Sage Reference Manual: 2D Graphics

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

CHAPTER

ONE

2D PLOTTING

Sage provides extensive 2D plotting functionality. The underlying rendering is done using the matplotlib Python library.

The following graphics primitives are supported:

- `arrow()` - an arrow from a min point to a max point.
- `circle()` - a circle with given radius
- `ellipse()` - an ellipse with given radii and angle
- `arc()` - an arc of a circle or an ellipse
- `disk()` - a filled disk (i.e. a sector or wedge of a circle)
- `line()` - a line determined by a sequence of points (this need not be straight!)
- `point()` - a point
- `text()` - some text
- `polygon()` - a filled polygon

The following plotting functions are supported:

- `plot()` - plot of a function or other Sage object (e.g., elliptic curve).
- `parametric_plot()`
- `implicit_plot()`
- `polar_plot()`
- `region_plot()`
- `list_plot()`
- `scatter_plot()`
- `bar_chart()`
- `contour_plot()`
- `density_plot()`
- `plot_vector_field()`
- `plot_slope_field()`
- `matrix_plot()`
- `complex_plot()`

- graphics_array()
- The following log plotting functions:
  - plot_loglog()
  - plot_semilogx() and plot_semilogy()
  - list_plot_loglog()
  - list_plot_semilogx() and list_plot_semilogy()

The following miscellaneous Graphics functions are included:

- Graphics()
- is_Graphics()
- hue()

Type ? after each primitive in Sage for help and examples.

EXAMPLES:

We draw a curve:

```
sage: plot(x^2, (x,0,5))
```

We draw a circle and a curve:

```
sage: circle((1,1), 1) + plot(x^2, (x,0,5))
```

Notice that the aspect ratio of the above plot makes the plot very tall because the plot adopts the default aspect ratio of the circle (to make the circle appear like a circle). We can change the aspect ratio to be what we normally expect for a plot by explicitly asking for an 'automatic' aspect ratio:

```
sage: show(circle((1,1), 1) + plot(x^2, (x,0,5)), aspect_ratio='automatic')
```

The aspect ratio describes the apparently height/width ratio of a unit square. If you want the vertical units to be twice as big as the horizontal units, specify an aspect ratio of 2:

```
sage: show(circle((1,1), 1) + plot(x^2, (x,0,5)), aspect_ratio=2)
```

The figsize option adjusts the figure size. The default figsize is 4. To make a figure that is roughly twice as big, use figsize=8:

```
sage: show(circle((1,1), 1) + plot(x^2, (x,0,5)), figsize=8)
```

You can also give separate horizontal and vertical dimensions:

```
sage: show(circle((1,1), 1) + plot(x^2, (x,0,5)), figsize=[4,8])
```

Note that the axes will not cross if the data is not on both sides of both axes, even if it is quite close:

```
sage: plot(x^3,(x,1,10))
```

When the labels have quite different orders of magnitude or are very large, scientific notation (the e notation for powers of ten) is used:

```
sage: plot(x^2,(x,480,500))  # no scientific notation
```

```
sage: plot(x^2,(x,300,500))  # scientific notation on y-axis
```

But you can fix your own tick labels, if you know what to expect and have a preference:

```
sage: plot(x^2,(x,300,500),ticks=[None,50000])
```

You can even have custom tick labels along with custom positioning.

```
sage: plot(x**2, (x,0,3), ticks=[[1,2.5],pi/2], tick_formatter=[["$x_1$","$x_2$"],pi]) # long time
```

We construct a plot involving several graphics objects:

```
sage: G = plot(cos(x), (x, -5, 5), thickness=5, color='green', title='A plot')
sage: P = polygon([[1,2], [5,6], [5,0]], color='red')
sage: G + P
```

Next we construct the reflection of the above polygon about the y-axis by iterating over the list of first-coordinates of the first graphic element of `P` (which is the actual Polygon; note that `P` is a Graphics object, which consists of a single polygon):

```
sage: Q = polygon([(-x,y) for x,y in P[0]], color='blue')
sage: Q   # show it
```

We combine together different graphics objects using "+":

```
sage: H = G + P + Q
sage: print H
Graphics object consisting of 3 graphics primitives
sage: type(H)
<class 'sage.plot.graphics.Graphics'>
sage: H[1]
Polygon defined by 3 points
sage: list(H[1])
[(1.0, 2.0), (5.0, 6.0), (5.0, 0.0)]
sage: H       # show it
```

We can put text in a graph:

```
sage: L = [[cos(pi*i/100)^3,sin(pi*i/100)] for i in range(200)]
sage: p = line(L, rgbcolor=(1/4,1/8,3/4))
sage: t = text('A Bulb', (1.5, 0.25))
sage: x = text('x axis', (1.5,-0.2))
sage: y = text('y axis', (0.4,0.9))
sage: g = p+t+x+y
sage: g.show(xmin=-1.5, xmax=2, ymin=-1, ymax=1)
```

We can add a title to a graph:

```
sage: x = var('x')
sage: plot(x^2, (x,-2,2), title='A plot of $x^2$')
```

We can set the position of the title:

```
sage: plot(x^2, (-2,2), title='Plot of $x^2$', title_pos=(0.5,-0.05))
```

We plot the Riemann zeta function along the critical line and see the first few zeros:

```
sage: i = CDF.0      # define i this way for maximum speed.
sage: p1 = plot(lambda t: arg(zeta(0.5+t*i)), 1,27,rgbcolor=(0.8,0,0))
sage: p2 = plot(lambda t: abs(zeta(0.5+t*i)), 1,27,color=hue(0.7))
sage: print p1 + p2
Graphics object consisting of 2 graphics primitives
sage: p1 + p2    # display it
```

Many concentric circles shrinking toward the origin:

```
sage: show(sum(circle((i,0), i, hue=sin(i/10)) for i in [10,9.9,..,0])) # long time
```

Here is a pretty graph:

```
sage: g = Graphics()
sage: for i in range(60):
...     p = polygon([(i*cos(i),i*sin(i)), (0,i), (i,0)],\
...                 color=hue(i/40+0.4), alpha=0.2)
...     g = g + p
...
sage: g.show(dpi=200, axes=False)
```

Another graph:

```
sage: x = var('x')
sage: P = plot(sin(x)/x, -4,4, color='blue') + \
...       plot(x*cos(x), -4,4, color='red') + \
...       plot(tan(x),-4,4, color='green')
...
sage: P.show(ymin=-pi,ymax=pi)
```

PYX EXAMPLES: These are some examples of plots similar to some of the plots in the PyX (http://pyx.sourceforge.net) documentation:

Symbolline:

```
sage: y(x) = x*sin(x^2)
sage: v = [(x, y(x)) for x in [-3,-2.95,..,3]]
sage: show(points(v, rgbcolor=(0.2,0.6, 0.1), pointsize=30) + plot(spline(v), -3.1, 3))
```

Cycliclink:

```
sage: x = var('x')
sage: g1 = plot(cos(20*x)*exp(-2*x), 0, 1)
sage: g2 = plot(2*exp(-30*x) - exp(-3*x), 0, 1)
sage: show(graphics_array([g1, g2], 2, 1), xmin=0)
```

Pi Axis:

```
sage: g1 = plot(sin(x), 0, 2*pi)
sage: g2 = plot(cos(x), 0, 2*pi, linestyle = "--")
sage: (g1+g2).show(ticks=pi/6, tick_formatter=pi)  # long time # show their sum, nicely formatted
```

An illustration of integration:

```
sage: f(x) = (x-3)*(x-5)*(x-7)+40
sage: P = line([(2,0),(2,f(2))], color='black')
sage: P += line([(8,0),(8,f(8))], color='black')
sage: P += polygon([(2,0),(2,f(2))] + [(x, f(x)) for x in [2,2.1,..,8]] + [(8,0),(2,0)],  rgbcolor=(
```

```
sage: P += text("$\\int_{a}^b f(x) dx$", (5, 20), fontsize=16, color='black')
sage: P += plot(f, (1, 8.5), thickness=3)
sage: P     # show the result
```

NUMERICAL PLOTTING:

Sage includes Matplotlib, which provides 2D plotting with an interface that is a likely very familiar to people doing numerical computation. For example,

```
sage: from pylab import *
sage: t = arange(0.0, 2.0, 0.01)
sage: s = sin(2*pi*t)
sage: P = plot(t, s, linewidth=1.0)
sage: xl = xlabel('time (s)')
sage: yl = ylabel('voltage (mV)')
sage: t = title('About as simple as it gets, folks')
sage: grid(True)
sage: savefig(os.path.join(SAGE_TMP, 'sage.png'))
```

We test that `imshow` works as well, verifying that trac ticket #2900 is fixed (in Matplotlib).

```
sage: imshow([[(0,0,0)]])
<matplotlib.image.AxesImage object at ...>
sage: savefig(os.path.join(SAGE_TMP, 'foo.png'))
```

Since the above overwrites many Sage plotting functions, we reset the state of Sage, so that the examples below work!

```
sage: reset()
```

See http://matplotlib.sourceforge.net for complete documentation about how to use Matplotlib.

TESTS: We test dumping and loading a plot.

```
sage: p = plot(sin(x), (x, 0,2*pi))
sage: Q = loads(dumps(p))
```

Verify that a clean sage startup does *not* import matplotlib:

```
sage: os.system("sage -c \"if 'matplotlib' in sys.modules: sys.exit(1)\"") # long time
0
```

AUTHORS:

- Alex Clemesha and William Stein (2006-04-10): initial version
- David Joyner: examples
- Alex Clemesha (2006-05-04) major update
- William Stein (2006-05-29): fine tuning, bug fixes, better server integration
- William Stein (2006-07-01): misc polish
- Alex Clemesha (2006-09-29): added contour_plot, frame axes, misc polishing
- Robert Miller (2006-10-30): tuning, NetworkX primitive
- Alex Clemesha (2006-11-25): added plot_vector_field, matrix_plot, arrow, bar_chart, Axes class usage (see axes.py)
- Bobby Moretti and William Stein (2008-01): Change plot to specify ranges using the (varname, min, max) notation.

- William Stein (2008-01-19): raised the documentation coverage from a miserable 12 percent to a 'wopping' 35 percent, and fixed and clarified numerous small issues.
- Jason Grout (2009-09-05): shifted axes and grid functionality over to matplotlib; fixed a number of smaller issues.
- Jason Grout (2010-10): rewrote aspect ratio portions of the code
- Jeroen Demeyer (2012-04-19): move parts of this file to graphics.py (trac ticket #12857)

sage.plot.plot.**SelectiveFormatter**(*formatter*, *skip_values*)

This matplotlib formatter selectively omits some tick values and passes the rest on to a specified formatter.

EXAMPLES:

This example is almost straight from a matplotlib example.

```
sage: from sage.plot.plot import SelectiveFormatter
sage: import matplotlib.pyplot as plt
sage: import numpy
sage: fig=plt.figure()
sage: ax=fig.add_subplot(111)
sage: t = numpy.arange(0.0, 2.0, 0.01)
sage: s = numpy.sin(2*numpy.pi*t)
sage: p = ax.plot(t, s)
sage: formatter=SelectiveFormatter(ax.xaxis.get_major_formatter(),skip_values=[0,1])
sage: ax.xaxis.set_major_formatter(formatter)
sage: fig.savefig(os.path.join(SAGE_TMP, 'test.png'))
```

sage.plot.plot.**adaptive_refinement**(*f*, *p1*, *p2*, *adaptive_tolerance=0.01*, *adaptive_recursion=5*, *level=0*)

The adaptive refinement algorithm for plotting a function `f`. See the docstring for plot for a description of the algorithm.

INPUT:

- `f` - a function of one variable
- `p1, p2` - two points to refine between
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large a relative difference should be before the adaptive refinement code considers it significant; see documentation for generate_plot_points for more information. See the documentation for `plot()` for more information on how the adaptive refinement algorithm works.

OUTPUT:

- `list` - a list of points to insert between `p1` and `p2` to get a better linear approximation between them

TESTS:

```
sage: from sage.plot.plot import adaptive_refinement
sage: adaptive_refinement(sin, (0,0), (pi,0), adaptive_tolerance=0.01, adaptive_recursion=0)
[]
sage: adaptive_refinement(sin, (0,0), (pi,0), adaptive_tolerance=0.01)
[(0.125*pi, 0.3826834323650898), (0.1875*pi, 0.5555702330196022), (0.25*pi, 0.7071067811865475),
```

This shows that lowering `adaptive_tolerance` and raising `adaptive_recursion` both increase the number of subdivision points, though which one creates more points is heavily dependent upon the function being plotted.

```
sage: x = var('x')
sage: f(x) = sin(1/x)
sage: n1 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_tolerance=0.01)); n1
15
sage: n2 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_recursion=10, adaptive_tolerance=0
79
sage: n3 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_tolerance=0.001)); n3
26
```

sage.plot.plot.**generate_plot_points**(*f*, *xrange*, *plot_points=5*, *adaptive_tolerance=0.01*, *adaptive_recursion=5*, *randomize=True*, *initial_points=None*)

Calculate plot points for a function f in the interval xrange. The adaptive refinement algorithm is also automatically invoked with a *relative* adaptive tolerance of adaptive_tolerance; see below.

INPUT:

- `f` - a function of one variable
- `p1, p2` - two points to refine between
- `plot_points` - (default: 5) the minimal number of plot points. (Note however that in any actual plot a number is passed to this, with default value 200.)
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large the relative difference should be before the adaptive refinement code considers it significant. If the actual difference is greater than adaptive_tolerance*delta, where delta is the initial subinterval size for the given xrange and plot_points, then the algorithm will consider it significant.
- `initial_points` - (default: None) a list of points that should be evaluated.

OUTPUT:

- a list of points (x, f(x)) in the interval xrange, which approximate the function f.

TESTS:

```
sage: from sage.plot.plot import generate_plot_points
sage: generate_plot_points(sin, (0, pi), plot_points=2, adaptive_recursion=0)
[(0.0, 0.0), (3.141592653589793, 1.2246...e-16)]

sage: from sage.plot.plot import generate_plot_points
sage: generate_plot_points(lambda x: x^2, (0, 6), plot_points=2, adaptive_recursion=0, initial_p
[(0.0, 0.0), (1.0, 1.0), (2.0, 4.0), (3.0, 9.0), (6.0, 36.0)]

sage: generate_plot_points(sin(x).function(x), (-pi, pi), randomize=False)
[(-3.141592653589793, -1.2246...e-16), (-2.748893571891069,
-0.3826834323650899), (-2.356194490192345, -0.707106781186547...),
(-2.1598449493429825, -0.831469612302545...), (-1.9634954084936207,
-0.9238795325112867), (-1.7671458676442586, -0.9807852804032304),
(-1.5707963267948966, -1.0), (-1.3744467859455345,
-0.9807852804032304), (-1.1780972450961724, -0.9238795325112867),
(-0.9817477042468103, -0.831469612302545...), (-0.7853981633974483,
-0.707106781186547...), (-0.39269908169872414, -0.3826834323650898),
(0.0, 0.0), (0.39269908169872414, 0.3826834323650898),
(0.7853981633974483, 0.707106781186547...), (0.9817477042468103,
0.831469612302545...), (1.1780972450961724, 0.9238795325112867),
(1.3744467859455345, 0.9807852804032304), (1.5707963267948966, 1.0),
(1.7671458676442586, 0.9807852804032304), (1.9634954084936207,
```

```
0.9238795325112867), (2.1598449493429825, 0.831469612302545...),
(2.356194490192345, 0.707106781186547...), (2.748893571891069,
0.3826834323650899), (3.141592653589793, 1.2246...e-16)]
```

This shows that lowering adaptive_tolerance and raising adaptive_recursion both increase the number of subdivision points. (Note that which creates more points is heavily dependent on the particular function plotted.)

```
sage: x = var('x')
sage: f(x) = sin(1/x)
sage: [len(generate_plot_points(f, (-pi, pi), plot_points=16, adaptive_tolerance=i, randomize=Fa
[97, 161, 275]

sage: [len(generate_plot_points(f, (-pi, pi), plot_points=16, adaptive_recursion=i, randomize=Fa
[97, 499, 2681]
```

sage.plot.plot.**graphics_array**(*array*, *n=None*, *m=None*)

graphics_array take a list of lists (or tuples) of graphics objects and plots them all on one canvas (single plot).

INPUT:

- array - a list of lists or tuples
- n, m - (optional) integers - if n and m are given then the input array is flattened and turned into an n x m array, with blank graphics objects padded at the end, if necessary.

EXAMPLE: Make some plots of sin functions:

```
sage: f(x) = sin(x)
sage: g(x) = sin(2*x)
sage: h(x) = sin(4*x)
sage: p1 = plot(f,(-2*pi,2*pi),color=hue(0.5)) # long time
sage: p2 = plot(g,(-2*pi,2*pi),color=hue(0.9)) # long time
sage: p3 = parametric_plot((f,g),(0,2*pi),color=hue(0.6)) # long time
sage: p4 = parametric_plot((f,h),(0,2*pi),color=hue(1.0)) # long time
```

Now make a graphics array out of the plots:

```
sage: graphics_array(((p1,p2),(p3,p4))) # long time
```

One can also name the array, and then use show() or save():

```
sage: ga = graphics_array(((p1,p2),(p3,p4))) # long time
sage: ga.show() # long time
```

Here we give only one row:

```
sage: p1 = plot(sin,(-4,4))
sage: p2 = plot(cos,(-4,4))
sage: g = graphics_array([p1, p2]); print g
Graphics Array of size 1 x 2
sage: g.show()
```

It is possible to use figsize to change the size of the plot as a whole:

```
sage: L = [plot(sin(k*x),(x,-pi,pi)) for k in [1..3]]
sage: G = graphics_array(L)
sage: G.show(figsize=[5,3])  # smallish and compact

sage: G.show(figsize=[10,20])  # bigger and tall and thin; long time (2s on sage.math, 2012)
```

```
sage: G.show(figsize=8)  # figure as a whole is a square
```

sage.plot.plot.**list_plot**(*data*, *plotjoined=False*, *aspect_ratio='automatic'*, ***kwargs*)

`list_plot` takes either a list of numbers, a list of tuples, a numpy array, or a dictionary and plots the corresponding points.

If given a list of numbers (that is, not a list of tuples or lists), `list_plot` forms a list of tuples `(i, x_i)` where `i` goes from 0 to `len(data)-1` and `x_i` is the `i`-th data value, and puts points at those tuple values.

`list_plot` will plot a list of complex numbers in the obvious way; any numbers for which `CC()` makes sense will work.

`list_plot` also takes a list of tuples `(x_i, y_i)` where `x_i` and `y_i` are the `i`-th values representing the `x`- and `y`-values, respectively.

If given a dictionary, `list_plot` interprets the keys as x-values and the values as y-values.

The `plotjoined=True` option tells `list_plot` to plot a line joining all the data.

It is possible to pass empty dictionaries, lists, or tuples to `list_plot`. Doing so will plot nothing (returning an empty plot).

EXAMPLES:

```
sage: list_plot([i^2 for i in range(5)]) # long time
```

Here are a bunch of random red points:

```
sage: r = [(random(),random()) for _ in range(20)]
sage: list_plot(r,color='red')
```

This gives all the random points joined in a purple line:

```
sage: list_plot(r, plotjoined=True, color='purple')
```

You can provide a numpy array.:

```
sage: import numpy
sage: list_plot(numpy.arange(10))

sage: list_plot(numpy.array([[1,2], [2,3], [3,4]]))
```

Plot a list of complex numbers:

```
sage: list_plot([1, I, pi + I/2, CC(.25, .25)])

sage: list_plot([exp(I*theta) for theta in [0, .2..pi]])
```

Note that if your list of complex numbers are all actually real, they get plotted as real values, so this

```
sage: list_plot([CDF(1), CDF(1/2), CDF(1/3)])
```

is the same as `list_plot([1, 1/2, 1/3])` – it produces a plot of the points $(0,1)$, $(1,1/2)$, and $(2,1/3)$.

If you have separate lists of x values and y values which you want to plot against each other, use the `zip` command to make a single list whose entries are pairs of (x,y) values, and feed the result into `list_plot`:

```
sage: x_coords = [cos(t)^3 for t in srange(0, 2*pi, 0.02)]
sage: y_coords = [sin(t)^3 for t in srange(0, 2*pi, 0.02)]
sage: list_plot(zip(x_coords, y_coords))
```

If instead you try to pass the two lists as separate arguments, you will get an error message:

```
sage: list_plot(x_coords, y_coords)
Traceback (most recent call last):
...
TypeError: The second argument 'plotjoined' should be boolean (True or False).  If you meant to
```

Dictionaries with numeric keys and values can be plotted:

```
sage: list_plot({22: 3365, 27: 3295, 37: 3135, 42: 3020, 47: 2880, 52: 2735, 57: 2550})
```

Plotting in logarithmic scale is possible for 2D list plots. There are two different syntaxes available:

```
sage: yl = [2**k for k in range(20)]
sage: list_plot(yl, scale='semilogy')  # long time  # log axis on vertical

sage: list_plot_semilogy(yl)        # same
```

Warning: If `plotjoined` is `False` then the axis that is in log scale must have all points strictly positive. For instance, the following plot will show no points in the figure since the points in the horizontal axis starts from $(0, 1)$.

```
sage: list_plot(yl, scale='loglog')          # both axes are log
```

Instead this will work. We drop the point $(0, 1)$.:

```
sage: list_plot(zip(range(1,len(yl)), yl[1:]), scale='loglog') # long time
```

We use `list_plot_loglog()` and plot in a different base.:

```
sage: list_plot_loglog(zip(range(1,len(yl)), yl[1:]), base=2) # long time
```

We can also change the scale of the axes in the graphics just before displaying:

```
sage: G = list_plot(yl) # long time
sage: G.show(scale=('semilogy', 2)) # long time
```

TESTS:

We check to see that the x/y min/max data are set correctly.

```
sage: d = list_plot([(100,100), (120, 120)]).get_minmax_data()
sage: d['xmin']
100.0
sage: d['ymin']
100.0
```

`sage.plot.plot.`**`list_plot_loglog`**(*data*, *plotjoined=False*, *base=10*, ***kwds*)

Plot the `data` in 'loglog' scale, that is, both the horizontal and the vertical axes will be in logarithmic scale.

INPUTS:

- `base` – (default: 10) the base of the logarithm. This must be greater than 1. The base can be also given as a list or tuple `(basex, basey)`. `basex` sets the base of the logarithm along the horizontal axis and `basey` sets the base along the vertical axis.

For all other inputs, look at the documentation of `list_plot()`.

EXAMPLES:

```
sage: yl = [5**k for k in range(10)]; xl = [2**k for k in range(10)]
sage: list_plot_loglog(zip(xl, yl)) # long time # plot in loglog scale with base 10

sage: list_plot_loglog(zip(xl, yl), base=2.1) # long time # with base 2.1 on both axes

sage: list_plot_loglog(zip(xl, yl), base=(2,5)) # long time
```

Warning: If `plotjoined` is `False` then the axis that is in log scale must have all points strictly positive. For instance, the following plot will show no points in the figure since the points in the horizontal axis starts from $(0, 1)$.

```
sage: yl = [2**k for k in range(20)]
sage: list_plot_loglog(yl)
```

Instead this will work. We drop the point $(0, 1)$.:

```
sage: list_plot_loglog(zip(range(1,len(yl)), yl[1:]))
```

sage.plot.plot.**list_plot_semilogx**(*data*, *plotjoined=False*, *base=10*, ***kwds*)

Plot `data` in 'semilogx' scale, that is, the horizontal axis will be in logarithmic scale.

INPUTS:

- `base` – (default: 10) the base of the logarithm. This must be greater than 1.

For all other inputs, look at the documentation of `list_plot()`.

EXAMPLES:

```
sage: yl = [2**k for k in range(12)]
sage: list_plot_semilogx(zip(yl,yl))
```

Warning: If `plotjoined` is `False` then the horizontal axis must have all points strictly positive. Otherwise the plot will come up empty. For instance the following plot contains a point at $(0, 1)$.

```
sage: yl = [2**k for k in range(12)]
sage: list_plot_semilogx(yl) # plot is empty because of '(0,1)'
```

We remove $(0, 1)$ to fix this.:

```
sage: list_plot_semilogx(zip(range(1, len(yl)), yl[1:]))
```

```
sage: list_plot_semilogx([(1,2),(3,4),(3,-1),(25,3)], base=2) # with base 2
```

sage.plot.plot.**list_plot_semilogy**(*data*, *plotjoined=False*, *base=10*, ***kwds*)

Plot `data` in 'semilogy' scale, that is, the vertical axis will be in logarithmic scale.

INPUTS:

- `base` – (default: 10) the base of the logarithm. This must be greater than 1.

For all other inputs, look at the documentation of `list_plot()`.

EXAMPLES:

```
sage: yl = [2**k for k in range(12)]
sage: list_plot_semilogy(yl) # plot in semilogy scale, base 10
```

Warning: If `plot joined` is `False` then the vertical axis must have all points strictly positive. Otherwise the plot will come up empty. For instance the following plot contains a point at $(1, 0)$.

```
sage: xl = [2**k for k in range(12)]; yl = range(len(xl))
sage: list_plot_semilogy(zip(xl,yl)) # plot empty due to (1,0)
```

We remove $(1, 0)$ to fix this.:

```
sage: list_plot_semilogy(zip(xl[1:],yl[1:]))
```

```
sage: list_plot_semilogy([2, 4, 6, 8, 16, 31], base=2) # with base 2
```

sage.plot.plot.**minmax_data**(*xdata*, *ydata*, *dict=False*)

Returns the minimums and maximums of xdata and ydata.

If dict is False, then minmax_data returns the tuple (xmin, xmax, ymin, ymax); otherwise, it returns a dictionary whose keys are 'xmin', 'xmax', 'ymin', and 'ymax' and whose values are the corresponding values.

EXAMPLES:

```
sage: from sage.plot.plot import minmax_data
sage: minmax_data([], [])
(-1, 1, -1, 1)
sage: minmax_data([-1, 2], [4, -3])
(-1, 2, -3, 4)
sage: d = minmax_data([-1, 2], [4, -3], dict=True)
sage: list(sorted(d.items()))
[('xmax', 2), ('xmin', -1), ('ymax', 4), ('ymin', -3)]
```

sage.plot.plot.**parametric_plot**(*funcs*, *aspect_ratio=1.0*, **args*, ***kwargs*)

Plot a parametric curve or surface in 2d or 3d.

`parametric_plot()` takes two or three functions as a list or a tuple and makes a plot with the first function giving the x coordinates, the second function giving the y coordinates, and the third function (if present) giving the z coordinates.

In the 2d case, `parametric_plot()` is equivalent to the `plot()` command with the option `parametric=True`. In the 3d case, `parametric_plot()` is equivalent to `parametric_plot3d()`. See each of these functions for more help and examples.

INPUT:

- `funcs` - 2 or 3-tuple of functions, or a vector of dimension 2 or 3.
- `other options` - passed to `plot()` or `parametric_plot3d()`

EXAMPLES: We draw some 2d parametric plots. Note that the default aspect ratio is 1, so that circles look like circles.

```
sage: t = var('t')
sage: parametric_plot( (cos(t), sin(t)), (t, 0, 2*pi))

sage: parametric_plot( (sin(t), sin(2*t)), (t, 0, 2*pi), color=hue(0.6) )

sage: parametric_plot((1, t), (t, 0, 4))
```

Note that in parametric_plot, there is only fill or no fill.

```
sage: parametric_plot((t, t^2), (t, -4, 4), fill = True)
```

A filled Hypotrochoid:

```
sage: parametric_plot([cos(x) + 2 * cos(x/4), sin(x) - 2 * sin(x/4)], (x,0, 8*pi), fill = True)

sage: parametric_plot( (5*cos(x), 5*sin(x), x), (x,-12, 12), plot_points=150, color="red") # lon

sage: y=var('y')
sage: parametric_plot( (5*cos(x), x*y, cos(x*y)), (x, -4,4), (y,-4,4)) # long time'

sage: t=var('t')
sage: parametric_plot( vector((sin(t), sin(2*t))), (t, 0, 2*pi), color='green') # long time
sage: parametric_plot( vector([t, t+1, t^2]), (t, 0, 1)) # long time
```

Plotting in logarithmic scale is possible with 2D plots. The keyword `aspect_ratio` will be ignored if the scale is not `'loglog'` or `'linear'`.:

```
sage: parametric_plot((x, x**2), (x, 1, 10), scale='loglog')
```

We can also change the scale of the axes in the graphics just before displaying. In this case, the `aspect_ratio` must be specified as `'automatic'` if the `scale` is set to `'semilogx'` or `'semilogy'`. For other values of the `scale` parameter, any `aspect_ratio` can be used, or the keyword need not be provided.:

```
sage: p = parametric_plot((x, x**2), (x, 1, 10))
sage: p.show(scale='semilogy', aspect_ratio='automatic')
```

TESTS:

```
sage: parametric_plot((x, t^2), (x, -4, 4))
Traceback (most recent call last):
...
ValueError: there are more variables than variable ranges

sage: parametric_plot((1, x+t), (x, -4, 4))
Traceback (most recent call last):
...
ValueError: there are more variables than variable ranges

sage: parametric_plot((-t, x+t), (x, -4, 4))
Traceback (most recent call last):
...
ValueError: there are more variables than variable ranges

sage: parametric_plot((1, x+t, y), (x, -4, 4), (t, -4, 4))
Traceback (most recent call last):
...
ValueError: there are more variables than variable ranges

sage: parametric_plot((1, x, y), 0, 4)
Traceback (most recent call last):
...
ValueError: there are more variables than variable ranges
```

sage.plot.plot.**plot**(*funcs*, *exclude=None*, *fillalpha=0.5*, *fillcolor='automatic'*, *detect_poles=False*, *plot_points=200*, *thickness=1*, *adaptive_tolerance=0.01*, *rgbcolor=(0, 0, 1)*, *adaptive_recursion=5*, *aspect_ratio='automatic'*, *alpha=1*, *legend_label=None*, *fill=False*, **args*, ***kwds*)

Use plot by writing

```
plot(X, ...)
```

where X is a Sage object (or list of Sage objects) that either is callable and returns numbers that can be coerced to floats, or has a plot method that returns a `GraphicPrimitive` object.

There are many other specialized 2D plot commands available in Sage, such as `plot_slope_field`, as well as various graphics primitives like `Arrow`; type `sage.plot.plot?` for a current list.

Type `plot.options` for a dictionary of the default options for plots. You can change this to change the defaults for all future plots. Use `plot.reset()` to reset to the default options.

PLOT OPTIONS:

- `plot_points` - (default: 200) the minimal number of plot points.
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large a difference should be before the adaptive refinement code considers it significant. See the documentation further below for more information, starting at "the algorithm used to insert".
- `base` - (default: 10) the base of the logarithm if a logarithmic scale is set. This must be greater than 1. The base can be also given as a list or tuple `(basex, basey)`. `basex` sets the base of the logarithm along the horizontal axis and `basey` sets the base along the vertical axis.
- `scale` – (default: `"linear"`) string. The scale of the axes. Possible values are `"linear"`, `"loglog"`, `"semilogx"`, `"semilogy"`.

  The scale can be also be given as single argument that is a list or tuple `(scale, base)` or `(scale, basex, basey)`.

  The `"loglog"` scale sets both the horizontal and vertical axes to logarithmic scale. The `"semilogx"` scale sets the horizontal axis to logarithmic scale. The `"semilogy"` scale sets the vertical axis to logarithmic scale. The `"linear"` scale is the default value when `Graphics` is initialized.
- `xmin` - starting x value
- `xmax` - ending x value
- `ymin` - starting y value in the rendered figure
- `ymax` - ending y value in the rendered figure
- `color` - an RGB tuple (r,g,b) with each of r,g,b between 0 and 1, or a color name as a string (e.g., 'purple'), or an HTML color such as '#aaff0b'.
- `detect_poles` - (Default: False) If set to True poles are detected. If set to "show" vertical asymptotes are drawn.
- `legend_color` - the color of the text for this item in the legend
- `legend_label` - the label for this item in the legend

Note:

- If the `scale` is `"linear"`, then irrespective of what `base` is set to, it will default to 10 and will remain unused.
- If you want to limit the plot along the horizontal axis in the final rendered figure, then pass the `xmin` and `xmax` keywords to the `show()` method. To limit the plot along the vertical axis, `ymin` and `ymax` keywords can be provided to either this `plot` command or to the `show` command.
- For the other keyword options that the `plot` function can take, refer to the method `show()`.

APPEARANCE OPTIONS:

The following options affect the appearance of the line through the points on the graph of X (these are the same as for the line function):

INPUT:

- `alpha` - How transparent the line is
- `thickness` - How thick the line is
- `rgbcolor` - The color as an RGB tuple
- `hue` - The color given as a hue

Any MATPLOTLIB line option may also be passed in. E.g.,

- **`linestyle` - (default: "-") The style of the line, which is one of**
  - `"-"` or `"solid"`
  - `"--"` or `"dashed"`
  - `"-."` or `"dash dot"`
  - `":"` or `"dotted"`
  - `"None"` or `" "` or `""` (nothing)

  The linestyle can also be prefixed with a drawing style (e.g., `"steps--"`)
  - `"default"` (connect the points with straight lines)
  - `"steps"` or `"steps-pre"` (step function; horizontal line is to the left of point)
  - `"steps-mid"` (step function; points are in the middle of horizontal lines)
  - `"steps-post"` (step function; horizontal line is to the right of point)
- **`marker` - The style of the markers, which is one of**
  - `"None"` or `" "` or `""` (nothing) – default
  - `","` (pixel), `"."` (point)
  - `"_"` (horizontal line), `"|"` (vertical line)
  - `"o"` (circle), `"p"` (pentagon), `"s"` (square), `"x"` (x), `"+"` (plus), `"*"` (star)
  - `"D"` (diamond), `"d"` (thin diamond)
  - `"H"` (hexagon), `"h"` (alternative hexagon)
  - `"<"` (triangle left), `">"` (triangle right), `"^"` (triangle up), `"v"` (triangle down)
  - `"1"` (tri down), `"2"` (tri up), `"3"` (tri left), `"4"` (tri right)
  - `0` (tick left), `1` (tick right), `2` (tick up), `3` (tick down)
  - `4` (caret left), `5` (caret right), `6` (caret up), `7` (caret down)
  - `"$...$"` (math TeX string)
- `markersize` - the size of the marker in points
- `markeredgecolor` – the color of the marker edge
- `markerfacecolor` – the color of the marker face
- `markeredgewidth` - the size of the marker edge in points

- `exclude` - (Default: None) values which are excluded from the plot range. Either a list of real numbers, or an equation in one variable.

FILLING OPTIONS:

- `fill` - (Default: False) One of:
  - "axis" or True: Fill the area between the function and the x-axis.
  - "min": Fill the area between the function and its minimal value.
  - "max": Fill the area between the function and its maximal value.
  - a number c: Fill the area between the function and the horizontal line y = c.
  - a function g: Fill the area between the function that is plotted and g.
  - a dictionary `d` (only if a list of functions are plotted): The keys of the dictionary should be integers. The value of `d[i]` specifies the fill options for the i-th function in the list. If `d[i] == [j]`: Fill the area between the i-th and the j-th function in the list. (But if `d[i] == j`: Fill the area between the i-th function in the list and the horizontal line y = j.)
- `fillcolor` - (default: 'automatic') The color of the fill. Either 'automatic' or a color.
- `fillalpha` - (default: 0.5) How transparent the fill is. A number between 0 and 1.

Note:

- this function does NOT simply sample equally spaced points between xmin and xmax. Instead it computes equally spaced points and adds small perturbations to them. This reduces the possibility of, e.g., sampling $\sin$ only at multiples of 2π, which would yield a very misleading graph.
- if there is a range of consecutive points where the function has no value, then those points will be excluded from the plot. See the example below on automatic exclusion of points.

EXAMPLES:

We plot the $\sin$ function:

```
sage: P = plot(sin, (0,10)); print P
Graphics object consisting of 1 graphics primitive
sage: len(P)      # number of graphics primitives
1
sage: len(P[0])   # how many points were computed (random)
225
sage: P           # render

sage: P = plot(sin, (0,10), plot_points=10); print P
Graphics object consisting of 1 graphics primitive
sage: len(P[0])   # random output
32
sage: P           # render
```

We plot with `randomize=False`, which makes the initial sample points evenly spaced (hence always the same). Adaptive plotting might insert other points, however, unless `adaptive_recursion=0`.

```
sage: p=plot(1, (x,0,3), plot_points=4, randomize=False, adaptive_recursion=0)
sage: list(p[0])
[(0.0, 1.0), (1.0, 1.0), (2.0, 1.0), (3.0, 1.0)]
```

Some colored functions:

```
sage: plot(sin, 0, 10, color='purple')
sage: plot(sin, 0, 10, color='#ff00ff')
```

We plot several functions together by passing a list of functions as input:

```
sage: plot([sin(n*x) for n in [1..4]], (0, pi))
```

We can also build a plot step by step from an empty plot:

```
sage: a = plot([]); a       # passing an empty list returns an empty plot (Graphics() object)
sage: a += plot(x**2); a    # append another plot
sage: a += plot(x**3); a    # append yet another plot
```

The function $\sin(1/x)$ wiggles wildly near 0. Sage adapts to this and plots extra points near the origin.

```
sage: plot(sin(1/x), (x, -1, 1))
```

Via the matplotlib library, Sage makes it easy to tell whether a graph is on both sides of both axes, as the axes only cross if the origin is actually part of the viewing area:

```
sage: plot(x^3,(x,0,2))  # this one has the origin
sage: plot(x^3,(x,1,2))  # this one does not
```

Another thing to be aware of with axis labeling is that when the labels have quite different orders of magnitude or are very large, scientific notation (the e notation for powers of ten) is used:

```
sage: plot(x^2,(x,480,500))  # this one has no scientific notation
sage: plot(x^2,(x,300,500))  # this one has scientific notation on y-axis
```

You can put a legend with `legend_label` (the legend is only put once in the case of multiple functions):

```
sage: plot(exp(x), 0, 2, legend_label='$e^x$')
```

Sage understands TeX, so these all are slightly different, and you can choose one based on your needs:

```
sage: plot(sin, legend_label='sin')
sage: plot(sin, legend_label='$sin$')
sage: plot(sin, legend_label='$\sin$')
```

It is possible to use a different color for the text of each label:

```
sage: p1 = plot(sin, legend_label='sin', legend_color='red')
sage: p2 = plot(cos, legend_label='cos', legend_color='green')
sage: p1 + p2
```

Note that the independent variable may be omitted if there is no ambiguity:

```
sage: plot(sin(1/x), (-1, 1))
```

Plotting in logarithmic scale is possible for 2D plots. There are two different syntaxes supported:

```
sage: plot(exp, (1, 10), scale='semilogy') # log axis on vertical

sage: plot_semilogy(exp, (1, 10)) # same thing

sage: plot_loglog(exp, (1, 10))   # both axes are log

sage: plot(exp, (1, 10), scale='loglog', base=2) # long time # base of log is 2
```

We can also change the scale of the axes in the graphics just before displaying:

```
sage: G = plot(exp, 1, 10) # long time
sage: G.show(scale=('semilogy', 2)) # long time
```

The algorithm used to insert extra points is actually pretty simple. On the picture drawn by the lines below:

```
sage: p = plot(x^2, (-0.5, 1.4)) + line([(0,0), (1,1)], color='green')
sage: p += line([(0.5, 0.5), (0.5, 0.5^2)], color='purple')
sage: p += point(((0, 0), (0.5, 0.5), (0.5, 0.5^2), (1, 1)), color='red', pointsize=20)
sage: p += text('A', (-0.05, 0.1), color='red')
sage: p += text('B', (1.01, 1.1), color='red')
sage: p += text('C', (0.48, 0.57), color='red')
sage: p += text('D', (0.53, 0.18), color='red')
sage: p.show(axes=False, xmin=-0.5, xmax=1.4, ymin=0, ymax=2)
```

You have the function (in blue) and its approximation (in green) passing through the points A and B. The algorithm finds the midpoint C of AB and computes the distance between C and D. If that distance exceeds the `adaptive_tolerance` threshold (*relative* to the size of the initial plot subintervals), the point D is added to the curve. If D is added to the curve, then the algorithm is applied recursively to the points A and D, and D and B. It is repeated `adaptive_recursion` times (5, by default).

The actual sample points are slightly randomized, so the above plots may look slightly different each time you draw them.

We draw the graph of an elliptic curve as the union of graphs of 2 functions.

```
sage: def h1(x): return abs(sqrt(x^3  - 1))
sage: def h2(x): return -abs(sqrt(x^3  - 1))
sage: P = plot([h1, h2], 1,4)
sage: P          # show the result
```

We can also directly plot the elliptic curve:

```
sage: E = EllipticCurve([0,-1])
sage: plot(E, (1, 4), color=hue(0.6))
```

We can change the line style as well:

```
sage: plot(sin(x), (x, 0, 10), linestyle='-.')
```

If we have an empty linestyle and specify a marker, we can see the points that are actually being plotted:

```
sage: plot(sin(x), (x,0,10), plot_points=20, linestyle='', marker='.')
```

The marker can be a TeX symbol as well:

```
sage: plot(sin(x), (x,0,10), plot_points=20, linestyle='', marker=r'$\checkmark$')
```

Sage currently ignores points that cannot be evaluated

```
sage: set_verbose(-1)
sage: plot(-x*log(x), (x,0,1))  # this works fine since the failed endpoint is just skipped.
sage: set_verbose(0)
```

This prints out a warning and plots where it can (we turn off the warning by setting the verbose mode temporarily to -1.)

```
sage: set_verbose(-1)
sage: plot(x^(1/3), (x,-1,1))
sage: set_verbose(0)
```

To plot the negative real cube root, use something like the following:

```
sage: plot(lambda x : RR(x).nth_root(3), (x,-1, 1))
```

Another way to avoid getting complex numbers for negative input is to calculate for the positive and negate the answer:

```
sage: plot(sign(x)*abs(x)^(1/3),-1,1)
```

We can detect the poles of a function:

```
sage: plot(gamma, (-3, 4), detect_poles = True).show(ymin = -5, ymax = 5)
```

We draw the Gamma-Function with its poles highlighted:

```
sage: plot(gamma, (-3, 4), detect_poles = 'show').show(ymin = -5, ymax = 5)
```

The basic options for filling a plot:

```
sage: p1 = plot(sin(x), -pi, pi, fill = 'axis')
sage: p2 = plot(sin(x), -pi, pi, fill = 'min')
sage: p3 = plot(sin(x), -pi, pi, fill = 'max')
sage: p4 = plot(sin(x), -pi, pi, fill = 0.5)
sage: graphics_array([[p1, p2], [p3, p4]]).show(frame=True, axes=False) # long time

sage: plot([sin(x), cos(2*x)*sin(4*x)], -pi, pi, fill = {0: 1}, fillcolor = 'red', fillalpha = 1
```

A example about the growth of prime numbers:

```
sage: plot(1.13*log(x), 1, 100, fill = lambda x: nth_prime(x)/floor(x), fillcolor = 'red')
```

Fill the area between a function and its asymptote:

```
sage: f = (2*x^3+2*x-1)/((x-2)*(x+1))
sage: plot([f, 2*x+2], -7,7, fill = {0: [1]}, fillcolor='#ccc').show(ymin=-20, ymax=20)
```

Fill the area between a list of functions and the x-axis:

```
sage: def b(n): return lambda x: bessel_J(n, x)
sage: plot([b(n) for n in [1..5]], 0, 20, fill = 'axis')
```

Note that to fill between the ith and jth functions, you must use dictionary key-value pairs `i:[j]`; key-value pairs like `i:j` will fill between the ith function and the line y=j:

```
sage: def b(n): return lambda x: bessel_J(n, x) + 0.5*(n-1)
sage: plot([b(c) for c in [1..5]], 0, 40, fill = dict([(i, [i+1]) for i in [0..3]]))
sage: plot([b(c) for c in [1..5]], 0, 40, fill = dict([(i, i+1) for i in [0..3]])) # long time
```

Extra options will get passed on to `show()`, as long as they are valid:

```
sage: plot(sin(x^2), (x, -3, 3), title='Plot of $\sin(x^2)$', axes_labels=['$x$','$y$']) # These
sage: plot(sin(x^2), (x, -3, 3), title='Plot of sin(x^2)', axes_labels=['x','y']) # These will n

sage: plot(sin(x^2), (x, -3, 3), figsize=[8,2])
sage: plot(sin(x^2), (x, -3, 3)).show(figsize=[8,2]) # These are equivalent
```

This includes options for custom ticks and formatting. See documentation for `show()` for more details.

```
sage: plot(sin(pi*x), (x, -8, 8), ticks=[[-7,-3,0,3,7],[-1/2,0,1/2]])
sage: plot(2*x+1,(x,0,5),ticks=[[0,1,e,pi,sqrt(20)],2],tick_formatter="latex")
```

This is particularly useful when setting custom ticks in multiples of pi.

```
sage: plot(sin(x),(x,0,2*pi),ticks=pi/3,tick_formatter=pi)
```

You can even have custom tick labels along with custom positioning.

```
sage: plot(x**2, (x,0,3), ticks=[[1,2.5],[0.5,1,2]], tick_formatter=[["$x_1$","$x_2$"],["$y_1$",
```

You can force Type 1 fonts in your figures by providing the relevant option as shown below. This also requires that LaTeX, dvipng and Ghostscript be installed:

```
sage: plot(x, typeset='type1') # optional - latex
```

A example with excluded values:

```
sage: plot(floor(x), (x, 1, 10), exclude = [1..10])
```

We exclude all points where `PrimePi` makes a jump:

```
sage: jumps = [n for n in [1..100] if prime_pi(n) != prime_pi(n-1)]
sage: plot(lambda x: prime_pi(x), (x, 1, 100), exclude = jumps)
```

Excluded points can also be given by an equation:

```
sage: g(x) = x^2-2*x-2
sage: plot(1/g(x), (x, -3, 4), exclude = g(x) == 0, ymin = -5, ymax = 5) # long time
```

`exclude` and `detect_poles` can be used together:

```
sage: f(x) = (floor(x)+0.5) / (1-(x-0.5)^2)
sage: plot(f, (x, -3.5, 3.5), detect_poles = 'show', exclude = [-3..3], ymin = -5, ymax = 5)
```

Regions in which the plot has no values are automatically excluded. The regions thus excluded are in addition to the exclusion points present in the `exclude` keyword argument.:

```
sage: set_verbose(-1)
sage: plot(arcsec, (x, -2, 2))  # [-1, 1] is excluded automatically

sage: plot(arcsec, (x, -2, 2), exclude=[1.5])  # x=1.5 is also excluded

sage: plot(arcsec(x/2), -2, 2)  # plot should be empty; no valid points

sage: plot(sqrt(x^2-1), -2, 2)  # [-1, 1] is excluded automatically

sage: plot(arccsc, -2, 2)       # [-1, 1] is excluded automatically

sage: set_verbose(0)
```

TESTS:

We do not randomize the endpoints:

```
sage: p = plot(x, (x,-1,1))
sage: p[0].xdata[0] == -1
True
sage: p[0].xdata[-1] == 1
True
```

We check to make sure that the x/y min/max data get set correctly when there are multiple functions.

```
sage: d = plot([sin(x), cos(x)], 100, 120).get_minmax_data()
sage: d['xmin']
100.0
```

```
sage: d['xmax']
120.0
```

We check various combinations of tuples and functions, ending with tests that lambda functions work properly with explicit variable declaration, without a tuple.

```
sage: p = plot(lambda x: x,(x,-1,1))
sage: p = plot(lambda x: x,-1,1)
sage: p = plot(x,x,-1,1)
sage: p = plot(x,-1,1)
sage: p = plot(x^2,x,-1,1)
sage: p = plot(x^2,xmin=-1,xmax=2)
sage: p = plot(lambda x: x,x,-1,1)
sage: p = plot(lambda x: x^2,x,-1,1)
sage: p = plot(lambda x: 1/x,x,-1,1)
sage: f(x) = sin(x+3)-.1*x^3
sage: p = plot(lambda x: f(x),x,-1,1)
```

We check to handle cases where the function gets evaluated at a point which causes an 'inf' or '-inf' result to be produced.

```
sage: p = plot(1/x, 0, 1)
sage: p = plot(-1/x, 0, 1)
```

Bad options now give better errors:

```
sage: P = plot(sin(1/x), (x,-1,3), foo=10)
Traceback (most recent call last):
...
RuntimeError: Error in line(): option 'foo' not valid.
sage: P = plot(x, (x,1,1)) # trac ticket #11753
Traceback (most recent call last):
...
ValueError: plot start point and end point must be different
```

We test that we can plot $f(x) = x$ (see trac ticket #10246):

```
sage: f(x)=x; f
x |--> x
sage: plot(f,(x,-1,1))
```

Check that trac ticket #15030 is fixed:

```
sage: plot(abs(log(x)), x)
```

Check that if excluded points are less than xmin then the exclusion still works for polar and parametric plots. The following should show two excluded points:

```
sage: set_verbose(-1)
sage: polar_plot(sin(sqrt(x^2-1)), (x,0,2*pi), exclude=[1/2,2,3])

sage: parametric_plot((sqrt(x^2-1),sqrt(x^2-1/2)), (x,0,5), exclude=[1,2,3])

sage: set_verbose(0)
```

sage.plot.plot.**plot_loglog**(*funcs*, *base=10*, **args*, ***kwds*)

Plot graphics in 'loglog' scale, that is, both the horizontal and the vertical axes will be in logarithmic scale.

INPUTS:

- base – (default: 10) the base of the logarithm. This must be greater than 1. The base can be also given as a list or tuple (basex, basey). basex sets the base of the logarithm along the horizontal axis and basey sets the base along the vertical axis.
- funcs – any Sage object which is acceptable to the plot().

For all other inputs, look at the documentation of plot().

EXAMPLES:

```
sage: plot_loglog(exp, (1,10)) # plot in loglog scale with base 10

sage: plot_loglog(exp, (1,10), base=2.1) # long time # with base 2.1 on both axes

sage: plot_loglog(exp, (1,10), base=(2,3))
```

sage.plot.plot.**plot_semilogx**(*funcs*, *base=10*, **args*, ***kwds*)

Plot graphics in 'semilogx' scale, that is, the horizontal axis will be in logarithmic scale.

INPUTS:

- base – (default: 10) the base of the logarithm. This must be greater than 1.
- funcs – any Sage object which is acceptable to the plot().

For all other inputs, look at the documentation of plot().

EXAMPLES:

```
sage: plot_semilogx(exp, (1,10)) # long time # plot in semilogx scale, base 10

sage: plot_semilogx(exp, (1,10), base=2) # with base 2
```

sage.plot.plot.**plot_semilogy**(*funcs*, *base=10*, **args*, ***kwds*)

Plot graphics in 'semilogy' scale, that is, the vertical axis will be in logarithmic scale.

INPUTS:

- base – (default: 10) the base of the logarithm. This must be greater than 1.
- funcs – any Sage object which is acceptable to the plot().

For all other inputs, look at the documentation of plot().

EXAMPLES:

```
sage: plot_semilogy(exp, (1,10)) # long time # plot in semilogy scale, base 10

sage: plot_semilogy(exp, (1,10), base=2) # long time # with base 2
```

sage.plot.plot.**polar_plot**(*funcs*, *aspect_ratio=1.0*, **args*, ***kwds*)

polar_plot takes a single function or a list or tuple of functions and plots them with polar coordinates in the given domain.

This function is equivalent to the plot() command with the options polar=True and aspect_ratio=1. For more help on options, see the documentation for plot().

INPUT:

- funcs - a function
- other options are passed to plot

EXAMPLES:

Here is a blue 8-leaved petal:

```
sage: polar_plot(sin(5*x)^2, (x, 0, 2*pi), color='blue')
```

A red figure-8:

```
sage: polar_plot(abs(sqrt(1 - sin(x)^2)), (x, 0, 2*pi), color='red')
```

A green limacon of Pascal:

```
sage: polar_plot(2 + 2*cos(x), (x, 0, 2*pi), color=hue(0.3))
```

Several polar plots:

```
sage: polar_plot([2*sin(x), 2*cos(x)], (x, 0, 2*pi))
```

A filled spiral:

```
sage: polar_plot(sqrt, 0, 2 * pi, fill = True)
```

Fill the area between two functions:

```
sage: polar_plot(cos(4*x) + 1.5, 0, 2*pi, fill=0.5 * cos(4*x) + 2.5, fillcolor='orange')
```

Fill the area between several spirals:

```
sage: polar_plot([(1.2+k*0.2)*log(x) for k in range(6)], 1, 3 * pi, fill = {0: [1], 2: [3], 4: [
```

Exclude points at discontinuities:

```
sage: polar_plot(log(floor(x)), (x, 1, 4*pi), exclude = [1..12])
```

sage.plot.plot.**reshape**(*v*, *n*, *m*)

Helper function for creating graphics arrays.

The input array is flattened and turned into an *nimesm* array, with blank graphics object padded at the end, if necessary.

INPUT:

- v - a list of lists or tuples
- n, m - integers

OUTPUT:

A list of lists of graphics objects

EXAMPLES:

```
sage: L = [plot(sin(k*x),(x,-pi,pi)) for k in range(10)]
sage: graphics_array(L,3,4)  # long time (up to 4s on sage.math, 2012)

sage: M = [[plot(sin(k*x),(x,-pi,pi)) for k in range(3)],[plot(cos(j*x),(x,-pi,pi)) for j in [3.
sage: graphics_array(M,6,1)  # long time (up to 4s on sage.math, 2012)
```

TESTS:

```
sage: L = [plot(sin(k*x),(x,-pi,pi)) for k in [1..3]]
sage: graphics_array(L,0,-1) # indirect doctest
Traceback (most recent call last):
...
AssertionError: array sizes must be positive
```

sage.plot.plot.**setup_for_eval_on_grid**(*v*, *xrange*, *yrange*, *plot_points*)

This function is deprecated. Please use `sage.plot.misc.setup_for_eval_on_grid` instead. Please note that that function has slightly different calling and return conventions which make it more generally applicable.

INPUT:

- `v` - a list of functions
- `xrange` - 2 or 3 tuple (if 3, first is a variable)
- `yrange` - 2 or 3 tuple
- `plot_points` - a positive integer

OUTPUT:

- `g` - tuple of fast callable functions
- `xstep` - step size in xdirection
- `ystep` - step size in ydirection
- `xrange` - tuple of 2 floats
- `yrange` - tuple of 2 floats

EXAMPLES:

```
sage: x,y = var('x,y')
sage: sage.plot.plot.setup_for_eval_on_grid([x^2 + y^2], (x,0,5), (y,0,pi), 11)
doctest:...: DeprecationWarning: sage.plot.plot.setup_for_eval_on_grid is deprecated.  Please us
See http://trac.sagemath.org/7008 for details.
([<sage.ext... object at ...>],
 0.5,
 0.3141592653589793,
 (0.0, 5.0),
 (0.0, 3.141592653589793))
```

We always plot at least two points; one at the beginning and one at the end of the ranges.

```
sage: sage.plot.plot.setup_for_eval_on_grid([x^2+y^2], (x,0,1), (y,-1,1), 1)
([<sage.ext... object at ...>],
1.0,
2.0,
(0.0, 1.0),
(-1.0, 1.0))
```

sage.plot.plot.**to_float_list**(*v*)

Given a list or tuple or iterable v, coerce each element of v to a float and make a list out of the result.

EXAMPLES:

```
sage: from sage.plot.plot import to_float_list
sage: to_float_list([1,1/2,3])
[1.0, 0.5, 3.0]
```

sage.plot.plot.**var_and_list_of_values**(*v*, *plot_points*)

INPUT:

- `v` - (v0, v1) or (var, v0, v1); if the former return the range of values between v0 and v1 taking plot_points steps; if var is given, also return var.
- `plot_points` - integer = 2 (the endpoints)

OUTPUT:

- var - a variable or None
- list - a list of floats

EXAMPLES:

```
sage: from sage.plot.plot import var_and_list_of_values
sage: var_and_list_of_values((var('theta'), 2, 5),  5)
doctest:...: DeprecationWarning: var_and_list_of_values is deprecated.  Please use sage.plot.mis
See http://trac.sagemath.org/7008 for details.
(theta, [2.0, 2.75, 3.5, 4.25, 5.0])
sage: var_and_list_of_values((2, 5),  5)
(None, [2.0, 2.75, 3.5, 4.25, 5.0])
sage: var_and_list_of_values((var('theta'), 2, 5),  2)
(theta, [2.0, 5.0])
sage: var_and_list_of_values((2, 5),  2)
(None, [2.0, 5.0])
```

sage.plot.plot.**xydata_from_point_list**(*points*)

Returns two lists (xdata, ydata), each coerced to a list of floats, which correspond to the x-coordinates and the y-coordinates of the points.

The points parameter can be a list of 2-tuples or some object that yields a list of one or two numbers.

This function can potentially be very slow for large point sets.

TESTS:

```
sage: from sage.plot.plot import xydata_from_point_list
sage: xydata_from_point_list([CC(0), CC(1)])   # ticket 8082
([0.0, 1.0], [0.0, 0.0])
```

This function should work for anything than can be turned into a list, such as iterators and such (see ticket #10478):

```
sage: xydata_from_point_list(iter([(0,0), (sqrt(3), 2)]))
([0.0, 1.7320508075688772], [0.0, 2.0])
sage: xydata_from_point_list((x, x^2) for x in range(5))
([0.0, 1.0, 2.0, 3.0, 4.0], [0.0, 1.0, 4.0, 9.0, 16.0])
sage: xydata_from_point_list(enumerate(prime_range(1, 15)))
([0.0, 1.0, 2.0, 3.0, 4.0, 5.0], [2.0, 3.0, 5.0, 7.0, 11.0, 13.0])
sage: from itertools import izip; xydata_from_point_list(izip([2,3,5,7], [11, 13, 17, 19]))
([2.0, 3.0, 5.0, 7.0], [11.0, 13.0, 17.0, 19.0])
```

CHAPTER

TWO

GRAPHICS OBJECTS

This file contains the definition of the classes `Graphics` and `GraphicsArray`. Usually, you don't create these classes directly (although you can do it), you would use `plot()` or `graphics_array()` instead.

AUTHORS:

- Jeroen Demeyer (2012-04-19): split off this file from plot.py (trac ticket #12857)
- Punarbasu Purkayastha (2012-05-20): Add logarithmic scale (trac ticket #4529)

class `sage.plot.graphics.`**Graphics**

Bases: `sage.structure.sage_object.SageObject`

The Graphics object is an empty list of graphics objects. It is useful to use this object when initializing a for loop where different graphics object will be added to the empty object.

EXAMPLES:

```
sage: G = Graphics(); print G
Graphics object consisting of 0 graphics primitives
sage: c = circle((1,1), 1)
sage: G+=c; print G
Graphics object consisting of 1 graphics primitive
```

Here we make a graphic of embedded isosceles triangles, coloring each one with a different color as we go:

```
sage: h=10; c=0.4; p=0.5;
sage: G = Graphics()
sage: for x in srange(1,h+1):
....:     l = [[0,x*sqrt(3)],[-x/2,-x*sqrt(3)/2],[x/2,-x*sqrt(3)/2],[0,x*sqrt(3)]]
....:     G+=line(l,color=hue(c + p*(x/h)))
sage: G.show(figsize=[5,5])
```

We can change the scale of the axes in the graphics before displaying.:

```
sage: G = plot(exp, 1, 10) # long time
sage: G.show(scale='semilogy') # long time
```

TESTS:

From trac ticket #4604, ensure Graphics can handle 3d objects:

```
sage: g = Graphics()
sage: g += sphere((1, 1, 1), 2)
sage: g.show()
```

We check that graphics can be pickled (we can't use equality on graphics so we just check that the load/dump cycle gives a `Graphics` instance):

```
sage: g = Graphics()
sage: g2 = loads(dumps(g))
sage: g2.show()

sage: isinstance(g2, Graphics)
True
```

add_primitive(*primitive*)

Adds a primitive to this graphics object.

EXAMPLES:

We give a very explicit example:

```
sage: G = Graphics()
sage: from sage.plot.line import Line
sage: from sage.plot.arrow import Arrow
sage: L = Line([3,4,2,7,-2],[1,2,e,4,5.],{'alpha':1,'thickness':2,'rgbcolor':(0,1,1),'legend
sage: A = Arrow(2,-5,.1,.2,{'width':3,'head':0,'rgbcolor':(1,0,0),'linestyle':'dashed','zord
sage: G.add_primitive(L)
sage: G.add_primitive(A)
sage: G
```

aspect_ratio()

Get the current aspect ratio, which is the ratio of height to width of a unit square, or 'automatic'.

OUTPUT: a positive float (height/width of a unit square), or 'automatic' (expand to fill the figure).

EXAMPLES:

The default aspect ratio for a new blank Graphics object is 'automatic':

```
sage: P = Graphics()
sage: P.aspect_ratio()
'automatic'
```

The aspect ratio can be explicitly set different than the object's default:

```
sage: P = circle((1,1), 1)
sage: P.aspect_ratio()
1.0
sage: P.set_aspect_ratio(2)
sage: P.aspect_ratio()
2.0
sage: P.set_aspect_ratio('automatic')
sage: P.aspect_ratio()
'automatic'
```

axes(*show=None*)

Set whether or not the x and y axes are shown by default.

INPUT:

- `show` - bool

If called with no input, return the current axes setting.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
```

By default the axes are displayed.

```
sage: L.axes()
True
```

But we turn them off, and verify that they are off

```
sage: L.axes(False)
sage: L.axes()
False
```

Displaying L now shows a triangle but no axes.

```
sage: L
```

axes_color(*c=None*)

Set the axes color.

If called with no input, return the current axes_color setting.

INPUT:

- c - an RGB color 3-tuple, where each tuple entry is a float between 0 and 1

EXAMPLES: We create a line, which has like everything a default axes color of black.

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.axes_color()
(0, 0, 0)
```

We change the axes color to red and verify the change.

```
sage: L.axes_color((1,0,0))
sage: L.axes_color()
(1.0, 0.0, 0.0)
```

When we display the plot, we'll see a blue triangle and bright red axes.

```
sage: L
```

axes_label_color(*c=None*)

Set the color of the axes labels.

The axes labels are placed at the edge of the x and y axes, and are not on by default (use the `axes_labels` command to set them; see the example below). This function just changes their color.

INPUT:

- c - an RGB 3-tuple of numbers between 0 and 1

If called with no input, return the current axes_label_color setting.

EXAMPLES: We create a plot, which by default has axes label color black.

```
sage: p = plot(sin, (-1,1))
sage: p.axes_label_color()
(0, 0, 0)
```

We change the labels to be red, and confirm this:

```
sage: p.axes_label_color((1,0,0))
sage: p.axes_label_color()
(1.0, 0.0, 0.0)
```

We set labels, since otherwise we won't see anything.

```
sage: p.axes_labels(['$x$ axis', '$y$ axis'])
```

In the plot below, notice that the labels are red:

```
sage: p
```

axes_labels(*l=None*)

Set the axes labels.

INPUT:

- `l` - (default: None) a list of two strings or None

OUTPUT: a 2-tuple of strings

If l is None, returns the current `axes_labels`, which is itself by default None. The default labels are both empty.

EXAMPLES: We create a plot and put x and y axes labels on it.

```
sage: p = plot(sin(x), (x, 0, 10))
sage: p.axes_labels(['$x$','$y$'])
sage: p.axes_labels()
('$x$', '$y$')
```

Now when you plot p, you see x and y axes labels:

```
sage: p
```

Notice that some may prefer axes labels which are not typeset:

```
sage: plot(sin(x), (x, 0, 10), axes_labels=['x','y'])
```

TESTS:

Unicode strings are acceptable; see trac ticket #13161. Note that this does not guarantee that matplotlib will handle the strings properly, although it should.

```
sage: c = circle((0,0), 1)
sage: c.axes_labels(['axe des abscisses', u'axe des ordonnées'])
sage: c._axes_labels
('axe des abscisses', u'axe des ordonnées')
```

axes_range(*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*)

Set the ranges of the x and y axes.

INPUT:

- `xmin, xmax, ymin, ymax` - floats

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.set_axes_range(-1, 20, 0, 2)
sage: d = L.get_axes_range()
sage: d['xmin'], d['xmax'], d['ymin'], d['ymax']
(-1.0, 20.0, 0.0, 2.0)
```

axes_width(*w=None*)

Set the axes width. Use this to draw a plot with really fat or really thin axes.

INPUT:

- `w` - a float

If called with no input, return the current `axes_width` setting.

EXAMPLE: We create a plot, see the default axes width (with funny Python float rounding), then reset the width to 10 (very fat).

```
sage: p = plot(cos, (-3,3))
sage: p.axes_width()
0.8
sage: p.axes_width(10)
sage: p.axes_width()
10.0
```

Finally we plot the result, which is a graph with very fat axes.

```
sage: p
```

description()

Print a textual description to stdout.

This method is mostly used for doctests.

EXAMPLES:

```
sage: print polytopes.n_cube(2).plot().description()
Polygon defined by 4 points: [(1.0, 1.0), (-1.0, 1.0), (-1.0, -1.0), (1.0, -1.0)]
Line defined by 2 points: [(-1.0, -1.0), (-1.0, 1.0)]
Line defined by 2 points: [(-1.0, -1.0), (1.0, -1.0)]
Line defined by 2 points: [(-1.0, 1.0), (1.0, 1.0)]
Line defined by 2 points: [(1.0, -1.0), (1.0, 1.0)]
Point set defined by 4 point(s): [(-1.0, -1.0), (-1.0, 1.0), (1.0, -1.0), (1.0, 1.0)]
```

fontsize(*s=None*)

Set the font size of axes labels and tick marks.

INPUT:

- `s` - integer, a font size in points.

If called with no input, return the current fontsize.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.fontsize()
10
sage: L.fontsize(20)
sage: L.fontsize()
20
```

All the numbers on the axes will be very large in this plot:

```
sage: L
```

get_axes_range()

Returns a dictionary of the range of the axes for this graphics object. This is fall back to the ranges in get_minmax_data() for any value which the user has not explicitly set.

> **Warning:** Changing the dictionary returned by this function does not change the axes range for this object. To do that, use the `set_axes_range()` method.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: list(sorted(L.get_axes_range().items()))
[('xmax', 3.0), ('xmin', 1.0), ('ymax', 5.0), ('ymin', -4.0)]
sage: L.set_axes_range(xmin=-1)
sage: list(sorted(L.get_axes_range().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 5.0), ('ymin', -4.0)]
```

get_minmax_data()

Return a dictionary whose keys give the xmin, xmax, ymin, and ymax data for this graphic.

> **Warning:** The returned dictionary is mutable, but changing it does not change the xmin/xmax/ymin/ymax data. The minmax data is a function of the primitives which make up this Graphics object. To change the range of the axes, call methods `xmin()`, `xmax()`, `ymin()`, `ymax()`, or `set_axes_range()`.

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: list(sorted(g.get_minmax_data().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 2.0), ('ymin', 1.0)]
```

Note that changing ymax doesn't change the output of get_minmax_data:

```
sage: g.ymax(10)
sage: list(sorted(g.get_minmax_data().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 2.0), ('ymin', 1.0)]
```

legend(*show=None*)

Set whether or not the legend is shown by default.

INPUT:

- `show` - (default: None) a boolean

If called with no input, return the current legend setting.

EXAMPLES:

By default no legend is displayed:

```
sage: P = plot(sin)
sage: P.legend()
False
```

But if we put a label then the legend is shown:

```
sage: P = plot(sin, legend_label='sin')
sage: P.legend()
True
```

We can turn it on or off:

```
sage: P.legend(False)
sage: P.legend()
False
sage: P.legend(True)
sage: P # show with the legend
```

matplotlib(*filename=None*, *xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*, *figsize=None*, *figure=None*, *sub=None*, *axes=None*, *axes_labels=None*, *fontsize=None*, *frame=False*, *verify=True*, *aspect_ratio=None*, *gridlines=None*, *gridlinesstyle=None*, *vgridlinesstyle=None*, *hgridlinesstyle=None*, *show_legend=None*, *legend_options={}*, *axes_pad=None*, *ticks_integer=None*, *tick_formatter=None*, *ticks=None*, *title=None*, *title_pos=None*, *base=None*, *scale=None*, *typeset='default'*)

Return a matplotlib figure object representing the graphic

EXAMPLES:

```
sage: c = circle((1,1),1)
sage: print c.matplotlib()
Figure(640x480)
```

To obtain the first matplotlib axes object inside of the figure, you can do something like the following.

```
sage: p=plot(sin(x), (x, -2*pi, 2*pi))
sage: figure=p.matplotlib()
sage: axes=figure.axes[0]
```

For input parameters, see the documentation for the `show()` method (this function accepts all except the transparent argument).

TESTS:

We verify that trac ticket #10291 is fixed:

```
sage: p = plot(sin(x), (x, -2*pi, 2*pi))
sage: figure = p.matplotlib()
sage: axes_range = p.get_axes_range()
sage: figure = p.matplotlib()
sage: axes_range2 = p.get_axes_range()
sage: axes_range == axes_range2
True
```

We verify that legend options are properly handled (trac ticket #12960). First, we test with no options, and next with an incomplete set of options.:

```
sage: p = plot(x, legend_label='aha')
sage: p.legend(True)
sage: pm = p.matplotlib()
sage: pm = p.matplotlib(legend_options={'font_size':'small'})
```

The title should not overlap with the axes labels nor the frame in the following plot (see trac ticket #10512):

```
sage: plot(sin(x^2), (x, -3, 3), title='Plot of sin(x^2)', axes_labels=['x','y'],frame=True)
```

`typeset` must not be set to an arbitrary string:

```
sage: plot(x, typeset='garbage')
Traceback (most recent call last):
...
ValueError: typeset must be set to one of 'default', 'latex', or
'type1'; got 'garbage'.
```

We verify that numerical options are changed to float before saving (trac ticket #14741). By default, Sage 5.10 changes float objects to the *RealLiteral* type. The patch changes them to float before creating *matplotlib* objects.:

```
sage: f = lambda x, y : (abs(cos((x + I * y) ** 4)) - 1) # long time
sage: g = implicit_plot(f,(-4, 4),(-3, 3),linewidth=0.6) # long time
sage: gm = g.matplotlib() # long time # without the patch, this goes BOOM -- er, TypeError
```

plot(**args*, ***kwds*)

Draw a 2D plot of this graphics object, which just returns this object since this is already a 2D graphics object.

EXAMPLES:

```
sage: S = circle((0,0), 2)
sage: S.plot() is S
True
```

plot3d(*z=0*, ***kwds*)

Returns an embedding of this 2D plot into the xy-plane of 3D space, as a 3D plot object. An optional parameter z can be given to specify the z-coordinate.

EXAMPLES:

```
sage: sum([plot(z*sin(x), 0, 10).plot3d(z) for z in range(6)]) # long time
```

save(*filename=None*, *legend_loc='best'*, *legend_fancybox=False*, *legend_font_style='normal'*, *legend_font_size='medium'*, *legend_font_variant='normal'*, *legend_title=None*, *legend_handlelength=0.05*, *legend_markerscale=0.6*, *legend_numpoints=2*, *legend_labelspacing=0.02*, *legend_columnspacing=None*, *legend_font_weight='medium'*, *legend_handletextpad=0.5*, *legend_ncol=1*, *legend_borderaxespad=None*, *legend_shadow=False*, *legend_borderpad=0.6*, *legend_font_family='sans-serif'*, *legend_back_color=(0.9, 0.9, 0.9)*, ***kwds*)

Save the graphics to an image file.

INPUT:

- `filename` – a string (default: autogenerated), the filename and the image format given by the extension, which can be one of the following:
  - `.eps`,
  - `.pdf`,
  - `.png`,
  - `.ps`,
  - `.sobj` (for a Sage object you can load later),
  - `.svg`,
  - empty extension will be treated as `.sobj`.

All other keyword arguments will be passed to the plotter.

OUTPUT:

- none.

EXAMPLES:

```
sage: c = circle((1,1), 1, color='red')
sage: filename = os.path.join(SAGE_TMP, 'test.png')
sage: c.save(filename, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

To make a figure bigger or smaller, use `figsize`:

```
sage: c.save(filename, figsize=5, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

By default, the figure grows to include all of the graphics and text, so the final image may not be exactly the figure size you specified. If you want a figure to be exactly a certain size, specify the keyword `fig_tight=False`:

```
sage: c.save(filename, figsize=[8,4], fig_tight=False,
....:        xmin=-1, xmax=3, ymin=-1, ymax=3)
```

You can also pass extra options to the plot command instead of this method, e.g.

```
sage: plot(x^2 - 5, (x, 0, 5), ymin=0).save(tmp_filename(ext='.png'))
```

will save the same plot as the one shown by this command:

```
sage: plot(x^2 - 5, (x, 0, 5), ymin=0)
```

(This test verifies that trac ticket #8632 is fixed.)

TESTS:

Legend labels should save correctly:

```
sage: P = plot(x,(x,0,1),legend_label='$xyz$')
sage: P.set_legend_options(back_color=(1,0,0))
sage: P.set_legend_options(loc=7)
sage: filename=os.path.join(SAGE_TMP, 'test.png')
sage: P.save(filename)
```

This plot should save with the frame shown, showing trac ticket #7524 is fixed (same issue as trac ticket #7981 and trac ticket #8632):

```
sage: var('x,y')
(x, y)
sage: a = plot_vector_field((x,-y),(x,-1,1),(y,-1,1))
sage: filename=os.path.join(SAGE_TMP, 'test2.png')
sage: a.save(filename)
```

The following plot should show the axes; fixes trac ticket #14782

```
sage: plot(x^2, (x, 1, 2), ticks=[[], []])
```

save_image(*filename=None*, **args*, ***kwds*)

Save an image representation of self. The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`. Currently this is implemented by calling the `save()` method of self, passing along all arguments and keywords.

Note: Not all image types are necessarily implemented for all graphics types. See `save()` for more details.

EXAMPLES:

```
sage: c = circle((1,1), 1, color='red')
sage: filename = os.path.join(SAGE_TMP, 'test.png')
sage: c.save_image(filename, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

set_aspect_ratio(*ratio*)

Set the aspect ratio, which is the ratio of height and width of a unit square (i.e., height/width of a unit square), or 'automatic' (expand to fill the figure).

INPUT:

- `ratio` - a positive real number or 'automatic'

EXAMPLES: We create a plot of the upper half of a circle, but it doesn't look round because the aspect ratio is off:

```
sage: P = plot(sqrt(1-x^2),(x,-1,1)); P
```

So we set the aspect ratio and now it is round:

```
sage: P.set_aspect_ratio(1)
sage: P.aspect_ratio()
1.0
sage: P
```

Note that the aspect ratio is inherited upon addition (which takes the max of aspect ratios of objects whose aspect ratio has been set):

```
sage: P + plot(sqrt(4-x^2),(x,-2,2))
```

In the following example, both plots produce a circle that looks twice as tall as wide:

```
sage: Q = circle((0,0), 0.5); Q.set_aspect_ratio(2)
sage: (P + Q).aspect_ratio(); P+Q
2.0
sage: (Q + P).aspect_ratio(); Q+P
2.0
```

set_axes_range(*xmin=None*, *xmax=None*, *ymin=None*, *ymax=None*)

Set the ranges of the x and y axes.

INPUT:

- `xmin, xmax, ymin, ymax` - floats

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.set_axes_range(-1, 20, 0, 2)
sage: d = L.get_axes_range()
sage: d['xmin'], d['xmax'], d['ymin'], d['ymax']
(-1.0, 20.0, 0.0, 2.0)
```

set_legend_options(***kwds*)

Set various legend options.

INPUT:

- `title` - (default: None) string, the legend title
- `ncol` - (default: 1) positive integer, the number of columns
- `columnspacing` - (default: None) the spacing between columns
- `borderaxespad` - (default: None) float, length between the axes and the legend
- `back_color` - (default: (0.9, 0.9, 0.9)) This parameter can be a string denoting a color or an RGB tuple. The string can be a color name as in ('red', 'green', 'yellow', ...) or a floating point number like '0.8' which gets expanded to (0.8, 0.8, 0.8). The tuple form is just a floating point RGB tuple with all values ranging from 0 to 1.
- `handlelength` - (default: 0.05) float, the length of the legend handles
- `handletextpad` - (default: 0.5) float, the pad between the legend handle and text
- `labelspacing` - (default: 0.02) float, vertical space between legend entries
- **`loc` - (default: 'best') May be a string, an integer or a tuple. String or** integer inputs must be one of the following:

– 0, ‘best’

– 1, ‘upper right’

– 2, ‘upper left’

– 3, ‘lower left’

– 4, ‘lower right’

– 5, ‘right’

– 6, ‘center left’

– 7, ‘center right’

– 8, ‘lower center’

– 9, ‘upper center’

– 10, ‘center’

– Tuple arguments represent an absolute (x, y) position on the plot in axes coordinates (meaning from 0 to 1 in each direction).

- `markerscale` - (default: 0.6) float, how much to scale the markers in the legend.
- `numpoints` - (default: 2) integer, the number of points in the legend for line
- `borderpad` - (default: 0.6) float, the fractional whitespace inside the legend border (between 0 and 1)
- `font_family` - (default: ‘sans-serif’) string, one of ‘serif’, ‘sans-serif’, ‘cursive’, ‘fantasy’, ‘monospace’
- `font_style` - (default: ‘normal’) string, one of ‘normal’, ‘italic’, ‘oblique’
- `font_variant` - (default: ‘normal’) string, one of ‘normal’, ‘small-caps’
- `font_weight` - (default: ‘medium’) string, one of ‘black’, ‘extra bold’, ‘bold’, ‘semibold’, ‘medium’, ‘normal’, ‘light’
- `font_size` - (default: ‘medium’) string, one of ‘xx-small’, ‘x-small’, ‘small’, ‘medium’, ‘large’, ‘x-large’, ‘xx-large’ or an absolute font size (e.g. 12)
- `shadow` - (default: False) boolean - draw a shadow behind the legend
- `fancybox` - (default: False) a boolean. If True, draws a frame with a round fancybox.

These are all keyword arguments.

OUTPUT: a dictionary of all current legend options

EXAMPLES:

By default, no options are set:

```
sage: p = plot(tan, legend_label='tan')
sage: p.set_legend_options()
{}
```

We build a legend with a shadow:

```
sage: p.set_legend_options(shadow=True)
sage: p.set_legend_options()['shadow']
True
```

To set the legend position to the center of the plot, all these methods are roughly equivalent:

```
sage: p.set_legend_options(loc='center'); p

sage: p.set_legend_options(loc=10); p

sage: p.set_legend_options(loc=(0.5,0.5)); p # aligns the bottom of the box to the center
```

show(*filename=None*, *linkmode=False*, *legend_loc='best'*, *legend_fancybox=False*, *legend_font_style='normal'*, *legend_font_size='medium'*, *legend_font_variant='normal'*, *legend_title=None*, *legend_handlelength=0.05*, *legend_markerscale=0.6*, *legend_numpoints=2*, *legend_labelspacing=0.02*, *legend_columnspacing=None*, *legend_font_weight='medium'*, *legend_handletextpad=0.5*, *legend_ncol=1*, *legend_borderaxespad=None*, *legend_shadow=False*, *legend_borderpad=0.6*, *legend_font_family='sans-serif'*, *legend_back_color=(0.9*, *0.9*, *0.9)*, ***kwds*)

Show this graphics image with the default image viewer.

OPTIONAL INPUT:

- `filename` - (default: None) string
- `dpi` - dots per inch
- `figsize` - [width, height]
- `fig_tight` - (default: True) whether to clip the drawing tightly around drawn objects. If True, then the resulting image will usually not have dimensions corresponding to `figsize`. If False, the resulting image will have dimensions corresponding to `figsize`.
- `aspect_ratio` - the perceived height divided by the perceived width. For example, if the aspect ratio is set to `1`, circles will look round and a unit square will appear to have sides of equal length, and if the aspect ratio is set `2`, vertical units will be twice as long as horizontal units, so a unit square will be twice as high as it is wide. If set to `'automatic'`, the aspect ratio is determined by `figsize` and the picture fills the figure.
- `axes` - (default: True)
- `axes_labels` - (default: None) list (or tuple) of two strings; the first is used as the label for the horizontal axis, and the second for the vertical axis.
- `fontsize` - (default: current setting – 10) positive integer; used for axes labels; if you make this very large, you may have to increase figsize to see all labels.
- `frame` - (default: False) draw a frame around the image
- `gridlines` - (default: None) can be any of the following:
  - None, False: do not add grid lines.
  - True, "automatic", "major": add grid lines at major ticks of the axes.
  - "minor": add grid at major and minor ticks.
  - [xlist,ylist]: a tuple or list containing two elements, where xlist (or ylist) can be any of the following.
    - None, False: don't add horizontal (or vertical) lines.
    - True, "automatic", "major": add horizontal (or vertical) grid lines at the major ticks of the axes.
    - "minor": add horizontal (or vertical) grid lines at major and minor ticks of axes.
    - an iterable yielding numbers n or pairs (n,opts), where n is the coordinate of the line and opt is a dictionary of MATPLOTLIB options for rendering the line.

•`gridlinesstyle, hgridlinesstyle, vgridlinesstyle` - (default: None) a dictionary of MATPLOTLIB options for the rendering of the grid lines, the horizontal grid lines or the vertical grid lines, respectively.

•**`linkmode` - (default: False) If True a string containing a link** to the produced file is returned.

•`transparent` - (default: False) If True, make the background transparent.

•`axes_pad` - (default: 0.02 on `"linear"` scale, 1 on `"log"` scale).

– In the `"linear"` scale, it determines the percentage of the axis range that is added to each end of each axis. This helps avoid problems like clipping lines because of line-width, etc. To get axes that are exactly the specified limits, set `axes_pad` to zero.

– On the `"log"` scale, it determines the exponent of the fraction of the minimum (resp. maximum) that is subtracted from the minimum (resp. added to the maximum) value of the axis. For instance if the minimum is m and the base of the axis is b then the new minimum after padding the axis will be $m - m/b^{\text{axes\_pad}}$.

•`ticks_integer` - (default: False) guarantee that the ticks are integers (the `ticks` option, if specified, will override this)

•`ticks` - A matplotlib locator for the major ticks, or a number. There are several options. For more information about locators, type `from matplotlib import ticker` and then `ticker?`.

– If this is a locator object, then it is the locator for the horizontal axis. A value of None means use the default locator.

– If it is a list of two locators, then the first is for the horizontal axis and one for the vertical axis. A value of None means use the default locator (so a value of [None, my_locator] uses my_locator for the vertical axis and the default for the horizontal axis).

– If in either case above one of the entries is a number m (something which can be coerced to a float), it will be replaced by a MultipleLocator which places major ticks at integer multiples of m. See examples.

– If in either case above one of the entries is a list of numbers, it will be replaced by a FixedLocator which places ticks at the locations specified. This includes the case of of the empty list, which will give no ticks. See examples.

•`tick_formatter` - A matplotlib formatter for the major ticks. There are several options. For more information about formatters, type `from matplotlib import ticker` and then `ticker?`.

If the value of this keyword is a single item, then this will give the formatting for the horizontal axis *only* (except for the `"latex"` option). If it is a list or tuple, the first is for the horizontal axis, the second for the vertical axis. The options are below:

– If one of the entries is a formatter object, then it used. A value of None means to use the default locator (so using `tick_formatter=[None, my_formatter]` uses my_formatter for the vertical axis and the default for the horizontal axis).

– If one of the entries is a symbolic constant such as π, e, or $sqrt(2)$, ticks will be formatted nicely at rational multiples of this constant.

Warning: This should only be used with the `ticks` option using nice rational multiples of that constant!

– If one of the entries is the string `"latex"`, then the formatting will be nice typesetting of the ticks. This is intended to be used when the tick locator for at least one of the axes is a list including some symbolic elements. This uses matplotlib's internal LaTeX rendering engine. If you want to use an external LaTeX compiler, then set the keyword option `typeset`. See examples.

- `title` - (default: None) The title for the plot
- **`title_pos` - (default: None) The position of the title for the** plot. It must be a tuple or a list of two real numbers `(x_pos, y_pos)` which indicate the relative position of the title within the plot. The plot itself can be considered to occupy, in relative terms, the region within a unit square $[0, 1]$ $times[0, 1]$. The title text is centered around the horizontal factor `x_pos` of the plot. The baseline of the title text is present at the vertical factor `y_pos` of the plot. Hence, `title_pos=(0.5, 0.5)` will center the title in the plot, whereas `title_pos=(0.5, 1.1)` will center the title along the horizontal direction, but will place the title a fraction 0.1 times above the plot.

  – If the first entry is a list of strings (or numbers), then the formatting for the horizontal axis will be typeset with the strings present in the list. Each entry of the list of strings must be provided with a corresponding number in the first entry of `ticks` to indicate its position on the axis. To typeset the strings with `"latex"` enclose them within `"$"` symbols. To have similar custom formatting of the labels along the vertical axis, the second entry must be a list of strings and the second entry of `ticks` must also be a list of numbers which give the positions of the labels. See the examples below.

- `show_legend` - (default: None) If True, show the legend
- **`legend_*` - all the options valid for `set_legend_options()`** prefixed with `legend_`
- `base` - (default: 10) the base of the logarithm if a logarithmic scale is set. This must be greater than 1. The base can be also given as a list or tuple `(basex, basey)`. `basex` sets the base of the logarithm along the horizontal axis and `basey` sets the base along the vertical axis.
- `scale` – (default: `"linear"`) string. The scale of the axes. Possible values are
  - `"linear"` – linear scaling of both the axes
  - `"loglog"` – sets both the horizontal and vertical axes to logarithmic scale
  - `"semilogx"` – sets only the horizontal axis to logarithmic scale.
  - `"semilogy"` – sets only the vertical axis to logarithmic scale.

  The scale can be also be given as single argument that is a list or tuple `(scale, base)` or `(scale, basex, basey)`.

Note:

– If the `scale` is `"linear"`, then irrespective of what `base` is set to, it will default to 10 and will remain unused.

- `typeset` – (default: `"default"`) string. The type of font rendering that should be used for the text. The possible values are
  - `"default"` – Uses matplotlib's internal text rendering engine called Mathtext (see http://matplotlib.org/users/mathtext.html). If you have modified the default matplotlib settings, for instance via a matplotlibrc file, then this option will not change any of those settings.
  - `"latex"` – LaTeX is used for rendering the fonts. This requires LaTeX, dvipng and Ghostscript to be installed.
  - `"type1"` – Type 1 fonts are used by matplotlib in the text in the figure. This requires LaTeX, dvipng and Ghostscript to be installed.

EXAMPLES:

```
sage: c = circle((1,1), 1, color='red')
sage: c.show(xmin=-1, xmax=3, ymin=-1, ymax=3)
```

You could also just make the picture larger by changing `figsize`:

```
sage: c.show(figsize=8, xmin=-1, xmax=3, ymin=-1, ymax=3)
```

You can turn off the drawing of the axes:

```
sage: show(plot(sin,-4,4), axes=False)
```

You can also label the axes. Putting something in dollar signs formats it as a mathematical expression:

```
sage: show(plot(sin,-4,4), axes_labels=('$x$','$y$'))
```

You can add a title to a plot:

```
sage: show(plot(sin,-4,4), title='A plot of $\sin(x)$')
```

You can also provide the position for the title to the plot. In the plot below the title is placed on the bottom left of the figure.:

```
sage: plot(sin, -4, 4, title='Plot sin(x)', title_pos=(0.05,-0.05))
```

If you want all the text to be rendered by using an external LaTeX installation then set the `typeset` to `"latex"`. This requires that LaTeX, dvipng and Ghostscript be installed:

```
sage: plot(x, typeset='latex') # optional - latex
```

If you want all the text in your plot to use Type 1 fonts, then set the `typeset` option to `"type1"`. This requires that LaTeX, dvipng and Ghostscript be installed:

```
sage: plot(x, typeset='type1') # optional - latex
```

You can turn on the drawing of a frame around the plots:

```
sage: show(plot(sin,-4,4), frame=True)
```

You can make the background transparent:

```
sage: plot(sin(x), (x, -4, 4), transparent=True)
```

We can change the scale of the axes in the graphics before displaying:

```
sage: G = plot(exp, 1, 10)
sage: G.show(scale='semilogy')
```

We can change the base of the logarithm too. The following changes the vertical axis to be on log scale, and with base 2. Note that the `base` argument will ignore any changes to the axis which is in linear scale.:

```
sage: G.show(scale='semilogy', base=2) # long time # y axis as powers of 2

sage: G.show(scale='semilogy', base=(3,2)) # base ignored for x-axis
```

The scale can be also given as a 2-tuple or a 3-tuple.:

```
sage: G.show(scale=('loglog', 2.1)) # long time # both x and y axes in base 2.1

sage: G.show(scale=('loglog', 2, 3)) # long time # x in base 2, y in base 3
```

The base need not be an integer, though it does have to be made a float.:

```
sage: G.show(scale='semilogx', base=float(e)) # base is e
```

Logarithmic scale can be used for various kinds of plots. Here are some examples.:

```
sage: G = list_plot(map(lambda i: 10**i, range(10))) # long time
sage: G.show(scale='semilogy') # long time

sage: G = parametric_plot((x, x**2), (x, 1, 10))
sage: G.show(scale='loglog')

sage: disk((5,5), 4, (0, 3*pi/2)).show(scale='loglog',base=2)

sage: x, y = var('x, y')
sage: G =  plot_vector_field((2^x,y^2),(x,1,10),(y,1,100))
sage: G.show(scale='semilogx',base=2)
```

Add grid lines at the major ticks of the axes.

```
sage: c = circle((0,0), 1)
sage: c.show(gridlines=True)
sage: c.show(gridlines="automatic")
sage: c.show(gridlines="major")
```

Add grid lines at the major and minor ticks of the axes.

```
sage: u,v = var('u v')
sage: f = exp(-(u^2+v^2))
sage: p = plot_vector_field(f.gradient(), (u,-2,2), (v,-2,2))
sage: p.show(gridlines="minor")
```

Add only horizontal or vertical grid lines.

```
sage: p = plot(sin,-10,20)
sage: p.show(gridlines=[None, "automatic"])
sage: p.show(gridlines=["minor", False])
```

Add grid lines at specific positions (using lists/tuples).

```
sage: x, y = var('x, y')
sage: p = implicit_plot((y^2-x^2)*(x-1)*(2*x-3)-4*(x^2+y^2-2*x)^2, \
....:              (x,-2,2), (y,-2,2), plot_points=1000)
sage: p.show(gridlines=[[1,0],[-1,0,1]])
```

Add grid lines at specific positions (using iterators).

```
sage: def maple_leaf(t):
....:     return (100/(100+(t-pi/2)^8))*(2-sin(7*t)-cos(30*t)/2)
sage: p = polar_plot(maple_leaf, -pi/4, 3*pi/2, color="red",plot_points=1000) # long time
sage: p.show(gridlines=( [-3,-2.75,..,3], xrange(-1,5,2) )) # long time
```

Add grid lines at specific positions (using functions).

```
sage: y = x^5 + 4*x^4 - 10*x^3 - 40*x^2 + 9*x + 36
sage: p = plot(y, -4.1, 1.1)
sage: xlines = lambda a,b: [z for z,m in y.roots()]
sage: p.show(gridlines=[xlines, [0]], frame=True, axes=False)
```

Change the style of all the grid lines.

```
sage: b = bar_chart([-3,5,-6,11], color='red')
sage: b.show(gridlines=([-1,-0.5,..,4],True),
....:     gridlinesstyle=dict(color="blue", linestyle=":"))
```

Change the style of the horizontal or vertical grid lines separately.

```
sage: p = polar_plot(2 + 2*cos(x), 0, 2*pi, color=hue(0.3))
sage: p.show(gridlines=True,
....:     hgridlinesstyle=dict(color="orange", linewidth=1.0),
....:     vgridlinesstyle=dict(color="blue", linestyle=":"))
```

Change the style of each grid line individually.

```
sage: x, y = var('x, y')
sage: p = implicit_plot((y^2-x^2)*(x-1)*(2*x-3)-4*(x^2+y^2-2*x)^2,
....:              (x,-2,2), (y,-2,2), plot_points=1000)
sage: p.show(gridlines=(
....:    [
....:     (1,{"color":"red","linestyle":":"}),
....:     (0,{"color":"blue","linestyle":"--"})
....:    ],
....:    [
....:     (-1,{"color":"red","linestyle":":"}),
....:     (0,{"color":"blue","linestyle":"--"}),
....:     (1,{"color":"red","linestyle":":"}),
....:    ]
....:    ),
....:    gridlinesstyle=dict(marker='x',color="black"))
```

Grid lines can be added to contour plots.

```
sage: f = sin(x^2 + y^2)*cos(x)*sin(y)
sage: c = contour_plot(f, (x, -4, 4), (y, -4, 4), plot_points=100)
sage: c.show(gridlines=True, gridlinesstyle={'linestyle':':','linewidth':1, 'color':'red'})
```

Grid lines can be added to matrix plots.

```
sage: M = MatrixSpace(QQ,10).random_element()
sage: matrix_plot(M).show(gridlines=True)
```

By default, Sage increases the horizontal and vertical axes limits by a certain percentage in all directions. This is controlled by the `axes_pad` parameter. Increasing the range of the axes helps avoid problems with lines and dots being clipped because the linewidth extends beyond the axes. To get axes limits that are exactly what is specified, set `axes_pad` to zero. Compare the following two examples

```
sage: plot(sin(x), (x, -pi, pi),thickness=2)+point((pi, -1), pointsize=15)
sage: plot(sin(x), (x, -pi, pi),thickness=2,axes_pad=0)+point((pi, -1), pointsize=15)
```

The behavior of the `axes_pad` parameter is different if the axis is in the `"log"` scale. If b is the base of the axis, the minimum value of the axis, is decreased by the factor $1/b^{\text{axes\_pad}}$ of the minimum and the maximum value of the axis is increased by the same factor of the maximum value. Compare the axes in the following two plots to see the difference.

```
sage: plot_loglog(x, (1.1*10**-2, 9990))

sage: plot_loglog(x, (1.1*10**-2, 9990), axes_pad=0)
```

Via matplotlib, Sage allows setting of custom ticks. See above for more details.

Here the labels are not so useful:

```
sage: plot(sin(pi*x), (x, -8, 8))
```

Now put ticks at multiples of 2:

```
sage: plot(sin(pi*x), (x, -8, 8), ticks=2)
```

Or just choose where you want the ticks:

```
sage: plot(sin(pi*x), (x, -8, 8), ticks=[[-7,-3,0,3,7],[-1/2,0,1/2]])
```

Or no ticks at all:

```
sage: plot(sin(pi*x), (x, -8, 8), ticks=[[],[]])
```

This can be very helpful in showing certain features of plots.

```
sage: plot(1.5/(1+e^(-x)), (x, -10, 10)) # doesn't quite show value of inflection point

sage: plot(1.5/(1+e^(-x)), (x, -10, 10), ticks=[None, 1.5/4]) # It's right at f(x)=0.75!
```

But be careful to leave enough room for at least two major ticks, so that the user can tell what the scale is:

```
sage: plot(x^2,(x,1,8),ticks=6).show()
Traceback (most recent call last):
...
ValueError: Expand the range of the independent variable to
allow two multiples of your tick locator (option 'ticks').
```

We can also do custom formatting if you need it. See above for full details:

```
sage: plot(2*x+1,(x,0,5),ticks=[[0,1,e,pi,sqrt(20)],2],tick_formatter="latex")
```

This is particularly useful when setting custom ticks in multiples of π.

```
sage: plot(sin(x),(x,0,2*pi),ticks=pi/3,tick_formatter=pi)
```

But keep in mind that you will get exactly the formatting you asked for if you specify both formatters. The first syntax is recommended for best style in that case.

```
sage: plot(arcsin(x),(x,-1,1),ticks=[None,pi/6],tick_formatter=["latex",pi]) # Nice-looking!

sage: plot(arcsin(x),(x,-1,1),ticks=[None,pi/6],tick_formatter=[None,pi]) # Not so nice-look
```

Custom tick labels can be provided by providing the keyword `tick_formatter` with the list of labels, and simultaneously providing the keyword `ticks` with the positions of the labels.

```
sage: plot(x, (x,0,3), ticks=[[1,2.5],[0.5,1,2]], tick_formatter=[["$x_1$","$x_2$"],["$y_1$"
```

The following sets the custom tick labels only along the horizontal axis.

```
sage: plot(x**2, (x,0,2), ticks=[[1,2], None], tick_formatter=[["$x_1$","$x_2$"], None])
```

If the number of tick labels do not match the number of positions of tick labels, then it results in an error.:

```
sage: plot(x**2, (x,0,2), ticks=[[2], None], tick_formatter=[["$x_1$","$x_2$"], None]).show(
Traceback (most recent call last):
...
ValueError: If the first component of the list 'tick_formatter' is a list then the first com
```

When using logarithmic scale along the axis, make sure to have enough room for two ticks so that the user can tell what the scale is. This can be effected by increasing the range of the independent variable, or by

changing the `base`, or by providing enough tick locations by using the `ticks` parameter.

By default, Sage will expand the variable range so that at least two ticks are included along the logarithmic axis. However, if you specify `ticks` manually, this safety measure can be defeated:

```
sage: list_plot_loglog([(1,2),(2,3)], plotjoined=True, ticks=[[1],[1]])
doctest:...: UserWarning: The x-axis contains fewer than
2 ticks; the logarithmic scale of the plot may not be apparent
to the reader.
doctest:...: UserWarning: The y-axis contains fewer than
2 ticks; the logarithmic scale of the plot may not be apparent
to the reader.
```

This one works, since the horizontal axis is automatically expanded to contain two ticks and the vertical axis is provided with two ticks:

```
sage: list_plot_loglog([(1,2),(2,3)], plotjoined=True, ticks=[None,[1,10]])
```

Another example in the log scale where both the axes are automatically expanded to show two major ticks:

```
sage: list_plot_loglog([(2,0.5), (3, 4)], plotjoined=True)
```

When using `title_pos`, it must be ensured that a list or a tuple of length two is used. Otherwise, an error is raised.:

```
sage; plot(x, -4, 4, title='Plot x', title_pos=0.05)
Traceback (most recent call last):
...
ValueError: 'title_pos' must be a list or tuple of two real numbers.
```

tick_label_color(*c=None*)

Set the color of the axes tick labels.

INPUT:

- `c` - an RGB 3-tuple of numbers between 0 and 1

If called with no input, return the current tick_label_color setting.

EXAMPLES:

```
sage: p = plot(cos, (-3,3))
sage: p.tick_label_color()
(0, 0, 0)
sage: p.tick_label_color((1,0,0))
sage: p.tick_label_color()
(1.0, 0.0, 0.0)
sage: p
```

xmax(*xmax=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.xmax()
3.0
sage: g.xmax(10)
sage: g.xmax()
10.0
```

xmin(*xmin=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.xmin()
-1.0
sage: g.xmin(-3)
sage: g.xmin()
-3.0
```

ymax(*ymax=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.ymax()
2.0
sage: g.ymax(10)
sage: g.ymax()
10.0
```

ymin(*ymin=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.ymin()
1.0
sage: g.ymin(-3)
sage: g.ymin()
-3.0
```

class sage.plot.graphics.**GraphicsArray**(*array*)

Bases: sage.structure.sage_object.SageObject

GraphicsArray takes a (m x n) list of lists of graphics objects and plots them all on one canvas.

append(*g*)

Appends a graphic to the array. Currently not implemented.

TESTS:

```
sage: from sage.plot.graphics import GraphicsArray
sage: G = GraphicsArray([plot(sin),plot(cos)])
sage: G.append(plot(tan))
Traceback (most recent call last):
...
NotImplementedError: Appending to a graphics array is not yet implemented
```

ncols()

Number of columns of the graphics array.

EXAMPLES:

```
sage: R = rainbow(6)
sage: L = [plot(x^n,(x,0,1),color=R[n]) for n in range(6)]
sage: G = graphics_array(L,2,3)
sage: G.ncols()
3
sage: graphics_array(L).ncols()
6
```

nrows()

Number of rows of the graphics array.

EXAMPLES:

```
sage: R = rainbow(6)
sage: L = [plot(x^n,(x,0,1),color=R[n]) for n in range(6)]
sage: G = graphics_array(L,2,3)
sage: G.nrows()
2
sage: graphics_array(L).nrows()
1
```

save (*filename=None*, *dpi=100*, *figsize=None*, *axes=None*, ***kwds*)
Save the `graphics_array` to a png called `filename`.

We loop over all graphics objects in the array and add them to a subplot and then render that.

INPUT:

- `filename` - (default: None) string
- `dpi` - dots per inch
- `figsize` - width or [width, height]
- `axes` - (default: True)

EXAMPLES:

```
sage: F = tmp_filename(ext='.png')
sage: L = [plot(sin(k*x),(x,-pi,pi)) for k in [1..3]]
sage: G = graphics_array(L)
sage: G.save(F, dpi=500, axes=False)  # long time (6s on sage.math, 2012)
```

TESTS:

```
sage: graphics_array([]).save()
sage: graphics_array([[]]).save()
```

save_image (*filename=None*, **args*, ***kwds*)
Save an image representation of self. The image type is determined by the extension of the filename. For example, this could be `.png`, `.jpg`, `.gif`, `.pdf`, `.svg`. Currently this is implemented by calling the `save()` method of self, passing along all arguments and keywords.

Note: Not all image types are necessarily implemented for all graphics types. See `save()` for more details.

EXAMPLES:

```
sage: plots = [[plot(m*cos(x + n*pi/4), (x,0, 2*pi)) for n in range(3)] for m in range(1,3)]
sage: G = graphics_array(plots)
sage: G.save_image(tmp_filename()+'.png')
```

show (*filename=None*, *dpi=100*, *figsize=None*, *axes=None*, ***kwds*)
Show this graphics array using the default viewer.

OPTIONAL INPUT:

- `filename` - (default: None) string
- `dpi` - dots per inch
- `figsize` - width or [width, height]
- `axes` - (default: True)

- `fontsize` - positive integer
- `frame` - (default: False) draw a frame around the image

EXAMPLES:

This draws a graphics array with four trig plots and no axes in any of the plots:

```
sage: G = graphics_array([[plot(sin), plot(cos)], [plot(tan), plot(sec)]])
sage: G.show(axes=False)
```

sage.plot.graphics.**is_Graphics**(*x*)

Return True if x is a Graphics object.

EXAMPLES:

```
sage: from sage.plot.graphics import is_Graphics
sage: is_Graphics(1)
False
sage: is_Graphics(disk((0.0, 0.0), 1, (0, pi/2)))
True
```

sage.plot.graphics.**show_default**(*default=None*)

Set the default for showing plots using any plot commands. If called with no arguments, returns the current default.

If this is `True` (the default) then any plot object when displayed will be displayed as an actual plot instead of text, i.e., the show command is not needed.

EXAMPLES:

The default starts out as `True` in interactive use and `False` in doctests:

```
sage: show_default()  # long time
doctest:...: DeprecationWarning: this is done automatically by the doctest framework
See http://trac.sagemath.org/14469 for details.
False
```

CHAPTER

THREE

ANIMATED PLOTS

Animations are generated from a list (or other iterable) of graphics objects. Images are produced by calling the `save_image` method on each input object, and using ImageMagick's `convert` program [IM] or `ffmpeg` [FF] to generate an animation. The output format is GIF by default, but can be any of the formats supported by `convert` or `ffmpeg`.

Warning: Note that ImageMagick and FFmpeg are not included with Sage, and must be installed by the user. On unix systems, type `which convert` at a command prompt to see if `convert` (part of the ImageMagick suite) is installed. If it is, you will be given its location. Similarly, you can check for `ffmpeg` with `which ffmpeg`. See [IM] or [FF] for installation instructions.

EXAMPLES:

The sine function:

```
sage: sines = [plot(c*sin(x), (-2*pi,2*pi), color=Color(c,0,0), ymin=-1,ymax=1) for c in sxrange(0,1,
sage: a = animate(sines)
sage: a
Animation with 5 frames
sage: a.show()  # optional -- ImageMagick
```

Animate using ffmpeg instead of ImageMagick:

```
sage: f = tmp_filename(ext='.gif')
sage: a.save(filename=f,use_ffmpeg=True) # optional -- ffmpeg
```

An animated `sage.plot.graphics.GraphicsArray` of rotating ellipses:

```
sage: E = animate((graphics_array([[ellipse((0,0),a,b,angle=t,xmin=-3,xmax=3)+circle((0,0),3,color='k
sage: E         # animations produced from a generator do not have a known length
Animation with unknown number of frames
sage: E.show()  # optional -- ImageMagick
```

A simple animation of a circle shooting up to the right:

```
sage: c = animate([circle((i,i), 1-1/(i+1), hue=i/10) for i in srange(0,2,0.2)],
....:             xmin=0,ymin=0,xmax=2,ymax=2,figsize=[2,2])
sage: c.show() # optional -- ImageMagick
```

Animations of 3d objects:

```
sage: var('s,t')
(s, t)
sage: def sphere_and_plane(x):
```

```
....:     return sphere((0,0,0),1,color='red',opacity=.5)+parametric_plot3d([t,x,s],(s,-1,1),(t,-1,1)
sage: sp = animate([sphere_and_plane(x) for x in sxrange(-1,1,.3)])
sage: sp[0]      # first frame
sage: sp[-1]     # last frame
sage: sp.show()  # optional -- ImageMagick

sage: (x,y,z) = var('x,y,z')
sage: def frame(t):
....:     return implicit_plot3d((x^2 + y^2 + z^2), (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=6
sage: a = animate([frame(t) for t in srange(.01,1.5,.2)])
sage: a[0]       # first frame
sage: a.show()   # optional -- ImageMagick
```

If the input objects do not have a `save_image` method, then the animation object attempts to make an image by calling its internal method `sage.plot.animate.Animation.make_image()`. This is illustrated by the following example:

```
sage: t = var('t')
sage: a = animate((sin(c*pi*t) for c in sxrange(1,2,.2)))
sage: a.show()  # optional -- ImageMagick
```

AUTHORS:

- William Stein
- John Palmieri
- Niles Johnson (2013-12): Expand to animate more graphics objects

REFERENCES:

class `sage.plot.animate.`**`Animation`**(*v=None*, ***kwds*)

Bases: `sage.structure.sage_object.SageObject`

Return an animation of a sequence of plots of objects.

INPUT:

- `v` - iterable of Sage objects. These should preferably be graphics objects, but if they aren't then `make_image()` is called on them.
- `xmin, xmax, ymin, ymax` - the ranges of the x and y axes.
- `**kwds` - all additional inputs are passed onto the rendering command. E.g., use figsize to adjust the resolution and aspect ratio.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.3)],
....:                xmin=0, xmax=2*pi, figsize=[2,1])
sage: a
Animation with 21 frames
sage: a[:5]
Animation with 5 frames
sage: a.show()          # optional -- ImageMagick
sage: a[:5].show()      # optional -- ImageMagick
```

The `show()` method takes arguments to specify the delay between frames (measured in hundredths of a second, default value 20) and the number of iterations (default value 0, which means to iterate forever). To iterate 4 times with half a second between each frame:

```
sage: a.show(delay=50, iterations=4) # optional -- ImageMagick
```

An animation of drawing a parabola:

```
sage: step = 0.1
sage: L = Graphics()
sage: v = []
sage: for i in srange(0,1,step):
....:       L += line([(i,i^2),(i+step,(i+step)^2)], rgbcolor=(1,0,0), thickness=2)
....:       v.append(L)
sage: a = animate(v, xmin=0, ymin=0)
sage: a.show() # optional -- ImageMagick
sage: show(L)
```

TESTS:

This illustrates that trac ticket #2066 is fixed (setting axes ranges when an endpoint is 0):

```
sage: animate([plot(sin, -1,1)], xmin=0, ymin=0)._kwds['xmin']
0
```

We check that trac ticket #7981 is fixed:

```
sage: a = animate([plot(sin(x + float(k)), (0, 2*pi), ymin=-5, ymax=5)
....:              for k in srange(0,2*pi,0.3)])
sage: a.show() # optional -- ImageMagick
```

Do not convert input iterator to a list:

```
sage: a = animate((x^p for p in sxrange(1,2,.1))); a
Animation with unknown number of frames
sage: a._frames
<generator object ...
```

ffmpeg(*savefile=None*, *show_path=False*, *output_format=None*, *ffmpeg_options=''*, *delay=None*, *iterations=0*, *pix_fmt='rgb24'*)

Returns a movie showing an animation composed from rendering the frames in self.

This method will only work if ffmpeg is installed. See http://www.ffmpeg.org for information about ffmpeg.

INPUT:

- `savefile` - file that the mpeg gets saved to.
- `show_path` - boolean (default: False); if True, print the path to the saved file
- `output_format` - string (default: None); format and suffix to use for the video. This may be 'mpg', 'mpeg', 'avi', 'gif', or any other format that ffmpeg can handle. If this is None and the user specifies `savefile` with a suffix, say `savefile='animation.avi'`, try to determine the format ('avi' in this case) from that file name. If no file is specified or if the suffix cannot be determined, 'mpg' is used.
- `ffmpeg_options` - string (default: ''); this string is passed directly to ffmpeg.
- `delay` - integer (default: None); delay in hundredths of a second between frames. The framerate is 100/delay. This is not supported for mpeg files: for mpegs, the frame rate is always 25 fps.
- `iterations` - integer (default: 0); number of iterations of animation. If 0, loop forever. This is only supported for animated gif output and requires ffmpeg version 0.9 or later. For older versions, set `iterations=None`.

- pix_fmt - string (default: 'rgb24'); used only for gif output. Different values such as 'rgb8' or 'pal8' may be necessary depending on how ffmpeg was installed. Set pix_fmt=None to disable this option.

If savefile is not specified: in notebook mode, display the animation; otherwise, save it to a default file name. Use sage.misc.misc.set_verbose() with level=1 to see additional output.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
....:                 xmin=0, xmax=2*pi, figsize=[2,1])
sage: dir = tmp_dir()
sage: a.ffmpeg(savefile=dir + 'new.mpg')        # optional -- ffmpeg
sage: a.ffmpeg(savefile=dir + 'new.avi')        # optional -- ffmpeg
sage: a.ffmpeg(savefile=dir + 'new.gif')        # optional -- ffmpeg
sage: a.ffmpeg(savefile=dir + 'new.mpg', show_path=True) # optional -- ffmpeg
Animation saved to .../new.mpg.
```

Note: If ffmpeg is not installed, you will get an error message like this:

```
Error: ffmpeg does not appear to be installed. Saving an animation to
a movie file in any format other than GIF requires this software, so
please install it and try again.

See www.ffmpeg.org for more information.
```

TESTS:

```
sage: a.ffmpeg(output_format='gif',delay=30,iterations=5)      # optional -- ffmpeg
```

gif(*delay=20*, *savefile=None*, *iterations=0*, *show_path=False*, *use_ffmpeg=False*)

Returns an animated gif composed from rendering the graphics objects in self.

This method will only work if either (a) the ImageMagick software suite is installed, i.e., you have the convert command or (b) ffmpeg is installed. See [IM] for more about ImageMagick, and see [FF] for more about ffmpeg. By default, this produces the gif using convert if it is present. If this can't find convert or if use_ffmpeg is True, then it uses ffmpeg instead.

INPUT:

- delay - (default: 20) delay in hundredths of a second between frames
- savefile - file that the animated gif gets saved to
- iterations - integer (default: 0); number of iterations of animation. If 0, loop forever.
- show_path - boolean (default: False); if True, print the path to the saved file
- use_ffmpeg - boolean (default: False); if True, use 'ffmpeg' by default instead of 'convert'.

If savefile is not specified: in notebook mode, display the animation; otherwise, save it to a default file name.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
....:                 xmin=0, xmax=2*pi, figsize=[2,1])
sage: dir = tmp_dir()
sage: a.gif()               # not tested
sage: a.gif(savefile=dir + 'my_animation.gif', delay=35, iterations=3)  # opticnal -- ImageM
sage: a.gif(savefile=dir + 'my_animation.gif', show_path=True) # optional -- ImageMagick
```

```
Animation saved to .../my_animation.gif.
sage: a.gif(savefile=dir + 'my_animation_2.gif', show_path=True, use_ffmpeg=True) # optional
Animation saved to .../my_animation_2.gif.
```

Note: If neither ffmpeg nor ImageMagick is installed, you will get an error message like this:

```
Error: Neither ImageMagick nor ffmpeg appears to be installed. Saving an
animation to a GIF file or displaying an animation requires one of these
packages, so please install one of them and try again.

See www.imagemagick.org and www.ffmpeg.org for more information.
```

---

graphics_array(*ncols=3*)

Return a sage.plot.graphics.GraphicsArray with plots of the frames of this animation, using the given number of columns. The frames must be acceptable inputs for sage.plot.graphics.GraphicsArray.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: v = [E.change_ring(GF(p)).plot(pointsize=30) for p in [97, 101, 103, 107]]
sage: a = animate(v, xmin=0, ymin=0)
sage: a
Animation with 4 frames
sage: a.show() # optional -- ImageMagick
```

Modify the default arrangement of array:

```
sage: g = a.graphics_array(); print g
Graphics Array of size 2 x 3
sage: g.show(figsize=[4,1]) # optional
```

Specify different arrangement of array and save with different file name:

```
sage: g = a.graphics_array(ncols=2); print g
Graphics Array of size 2 x 2
sage: f = tmp_filename(ext='.png')
sage: g.show(f) # optional
```

Frames can be specified as a generator too; it is internally converted to a list:

```
sage: t = var('t')
sage: b = animate((plot(sin(c*pi*t)) for c in sxrange(1,2,.2)))
sage: g = b.graphics_array(); print g
Graphics Array of size 2 x 3
sage: g.show() # optional
```

make_image(*frame*, *filename*, ***kwds*)

Given a frame which has no `save_image()` method, make a graphics object and save it as an image with the given filename. By default, this is sage.plot.plot.plot(). To make animations of other objects, override this method in a subclass.

EXAMPLES:

```
sage: from sage.plot.animate import Animation
sage: class MyAnimation(Animation):
....:     def make_image(self, frame, filename, **kwds):
....:         P = parametric_plot(frame[0], frame[1], **frame[2])
```

```
....:         P.save_image(filename,**kwds)

sage: t = var('t')
sage: x = lambda t: cos(t)
sage: y = lambda n,t: sin(t)/n
sage: B = MyAnimation([([x(t), y(i+1,t)],(t,0,1), {'color':Color((1,0,i/4)), 'aspect_ratio':

sage: d = B.png()
sage: v = os.listdir(d); v.sort(); v
['00000000.png', '00000001.png', '00000002.png', '00000003.png']
sage: B.show()  # not tested

sage: class MyAnimation(Animation):
....:     def make_image(self, frame, filename, **kwds):
....:         G = frame.plot()
....:         G.set_axes_range(floor(G.xmin()),ceil(G.xmax()),floor(G.ymin()),ceil(G.ymax()))
....:         G.save_image(filename, **kwds)

sage: B = MyAnimation([graphs.CompleteGraph(n) for n in range(7,11)], figsize=5)
sage: d = B.png()
sage: v = os.listdir(d); v.sort(); v
['00000000.png', '00000001.png', '00000002.png', '00000003.png']
sage: B.show()  # not tested
```

png (*dir=None*)

Render PNG images of the frames in this animation, saving them in `dir`. Return the absolute path to that directory. If the frames have been previously rendered and `dir` is `None`, just return the directory in which they are stored.

When `dir` is other than `None`, force re-rendering of frames.

INPUT:

- `dir` – Directory in which to store frames. Default `None`; in this case, a temporary directory will be created for storing the frames.

EXAMPLES:

```
sage: a = animate([plot(x^2 + n) for n in range(4)])
sage: d = a.png()
sage: v = os.listdir(d); v.sort(); v
['00000000.png', '00000001.png', '00000002.png', '00000003.png']
```

save (*filename=None*, *show_path=False*, *use_ffmpeg=False*)

Save this animation.

INPUT:

- `filename` - (default: None) name of save file
- `show_path` - boolean (default: False); if True, print the path to the saved file
- `use_ffmpeg` - boolean (default: False); if True, use 'ffmpeg' by default instead of 'convert' when creating GIF files.

If filename is None, then in notebook mode, display the animation; otherwise, save the animation to a GIF file. If filename ends in '.sobj', save to an sobj file. Otherwise, try to determine the format from the filename extension ('.mpg', '.gif', '.avi', etc.). If the format cannot be determined, default to GIF.

For GIF files, either ffmpeg or the ImageMagick suite must be installed. For other movie formats, ffmpeg must be installed. An sobj file can be saved with no extra software installed.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
....:                 xmin=0, xmax=2*pi, figsize=[2,1])
sage: dir = tmp_dir()
sage: a.save()          # not tested
sage: a.save(dir + 'wave.gif')    # optional -- ImageMagick
sage: a.save(dir + 'wave.gif', show_path=True)    # optional -- ImageMagick
Animation saved to file .../wave.gif.
sage: a.save(dir + 'wave.avi', show_path=True)    # optional -- ffmpeg
Animation saved to file .../wave.avi.
sage: a.save(dir + 'wave0.sobj')
sage: a.save(dir + 'wave1.sobj', show_path=True)
Animation saved to file .../wave1.sobj.
```

show (*delay=20*, *iterations=0*)

Show this animation.

INPUT:

- `delay` - (default: 20) delay in hundredths of a second between frames
- `iterations` - integer (default: 0); number of iterations of animation. If 0, loop forever.

Note: Currently this is done using an animated gif, though this could change in the future. This requires that either ffmpeg or the ImageMagick suite (in particular, the `convert` command) is installed.

See also the `ffmpeg()` method.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
....:                 xmin=0, xmax=2*pi, figsize=[2,1])
sage: a.show()        # optional -- ImageMagick
```

The preceding will loop the animation forever. If you want to show only three iterations instead:

```
sage: a.show(iterations=3)     # optional -- ImageMagick
```

To put a half-second delay between frames:

```
sage: a.show(delay=50)         # optional -- ImageMagick
```

Note: If you don't have ffmpeg or ImageMagick installed, you will get an error message like this:

```
Error: Neither ImageMagick nor ffmpeg appears to be installed. Saving an
animation to a GIF file or displaying an animation requires one of these
packages, so please install one of them and try again.

See www.imagemagick.org and www.ffmpeg.org for more information.
```

---

sage.plot.animate.**animate** (*frames*, ***kwds*)

Animate a list of frames by creating a `sage.plot.animate.Animation` object.

EXAMPLES:

```
sage: t = var('t')
sage: a = animate((cos(c*pi*t) for c in sxrange(1,2,.2)))
sage: a.show()  # optional -- ImageMagick
```

CHAPTER

FOUR

ARCS OF CIRCLES AND ELLIPSES

class sage.plot.arc.**Arc**(*x*, *y*, *r1*, *r2*, *angle*, *s1*, *s2*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the Arc graphics type. See `arc?` for information about actually plotting an arc of a circle or an ellipse.

INPUT:

- `x, y` - coordinates of the center of the arc
- `r1`, `r2` - lengths of the two radii
- `angle` - angle of the horizontal with width
- `sector` - sector of angle
- `options` - dict of valid plot options to pass to constructor

EXAMPLES:

Note that the construction should be done using `arc`:

```
sage: from sage.plot.arc import Arc
sage: print Arc(0,0,1,1,pi/4,pi/4,pi/2,{})
Arc with center (0.0,0.0) radii (1.0,1.0) angle 0.785398163397 inside the sector (0.785398163397
```

get_minmax_data()

Returns a dictionary with the bounding box data.

The bounding box is computed as minimal as possible.

EXAMPLES:

An example without angle:

```
sage: p = arc((-2, 3), 1, 2)
sage: d = p.get_minmax_data()
sage: d['xmin']
-3.0
sage: d['xmax']
-1.0
sage: d['ymin']
1.0
sage: d['ymax']
5.0
```

The same example with a rotation of angle $\pi/2$:

```
sage: p = arc((-2, 3), 1, 2, pi/2)
sage: d = p.get_minmax_data()
sage: d['xmin']
-4.0
sage: d['xmax']
0.0
sage: d['ymin']
2.0
sage: d['ymax']
4.0
```

plot3d()

TESTS:

```
sage: from sage.plot.arc import Arc
sage: Arc(0,0,1,1,0,0,1,{}).plot3d()
Traceback (most recent call last):
...
NotImplementedError
```

sage.plot.arc.**arc**(*center*, *r1*, *r2=None*, *angle=0.0*, *sector=(0.0, 6.283185307179586)*, *rgbcolor='blue'*, *thickness=1*, *zorder=5*, *aspect_ratio=1.0*, *alpha=1*, *linestyle='solid'*, ***options*)

An arc (that is a portion of a circle or an ellipse)

Type `arc.options` to see all options.

INPUT:

- `center` - 2-tuple of real numbers - position of the center.
- `r1`, `r2` - positive real numbers - radii of the ellipse. If only `r1` is set, then the two radii are supposed to be equal and this function returns an arc of of circle.
- `angle` - real number - angle between the horizontal and the axis that corresponds to `r1`.
- `sector` - 2-tuple (default: (0,2*pi))- angles sector in which the arc will be drawn.

OPTIONS:

- `alpha` - float (default: 1) - transparency
- `thickness` - float (default: 1) - thickness of the arc
- `color`, `rgbcolor` - string or 2-tuple (default: 'blue') - the color of the arc
- `linestyle` - string (default: `'solid'`) - The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.

EXAMPLES:

Plot an arc of circle centered at (0,0) with radius 1 in the sector $(\pi/4, 3*\pi/4)$:

```
sage: arc((0,0), 1, sector=(pi/4,3*pi/4))
```

Plot an arc of an ellipse between the angles 0 and $\pi/2$:

```
sage: arc((2,3), 2, 1, sector=(0,pi/2))
```

Plot an arc of a rotated ellipse between the angles 0 and $\pi/2$:

```
sage: arc((2,3), 2, 1, angle=pi/5, sector=(0,pi/2))
```

Plot an arc of an ellipse in red with a dashed linestyle:

```
sage: arc((0,0), 2, 1, 0, (0,pi/2), linestyle="dashed", color="red")
sage: arc((0,0), 2, 1, 0, (0,pi/2), linestyle="--", color="red")
```

The default aspect ratio for arcs is 1.0:

```
sage: arc((0,0), 1, sector=(pi/4,3*pi/4)).aspect_ratio()
1.0
```

It is not possible to draw arcs in 3D:

```
sage: A = arc((0,0,0), 1)
Traceback (most recent call last):
...
NotImplementedError
```

CHAPTER

FIVE

ARROWS

class sage.plot.arrow.**Arrow**(*xtail*, *ytail*, *xhead*, *yhead*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class that initializes the (line) arrow graphics type

EXAMPLES:

We create an arrow graphics object, then take the 0th entry in it to get the actual Arrow graphics primitive:

```
sage: P = arrow((0,1), (2,3))[0]
sage: type(P)
<class 'sage.plot.arrow.Arrow'>
sage: P
Arrow from (0.0,1.0) to (2.0,3.0)
```

get_minmax_data()

Returns a bounding box for this arrow.

EXAMPLES:

```
sage: d = arrow((1,1), (5,5)).get_minmax_data()
sage: d['xmin']
1.0
sage: d['xmax']
5.0
```

plot3d(*ztail=0*, *zhead=0*, ***kwds*)

Takes 2D plot and places it in 3D.

EXAMPLES:

```
sage: A = arrow((0,0),(1,1))[0].plot3d()
sage: A.jmol_repr(A.testing_render_params())[0]
'draw line_1 diameter 2 arrow {0.0 0.0 0.0}  {1.0 1.0 0.0} '
```

Note that we had to index the arrow to get the Arrow graphics primitive. We can also change the height via the plot3d method of Graphics, but only as a whole:

```
sage: A = arrow((0,0),(1,1)).plot3d(3)
sage: A.jmol_repr(A.testing_render_params())[0][0]
'draw line_1 diameter 2 arrow {0.0 0.0 3.0}  {1.0 1.0 3.0} '
```

Optional arguments place both the head and tail outside the xy-plane, but at different heights. This must be done on the graphics primitive obtained by indexing:

```
sage: A=arrow((0,0),(1,1))[0].plot3d(3,4)
sage: A.jmol_repr(A.testing_render_params())[0]
'draw line_1 diameter 2 arrow {0.0 0.0 3.0}  {1.0 1.0 4.0} '
```

class sage.plot.arrow.**CurveArrow**(*path*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Returns an arrow graphics primitive along the provided path (bezier curve).

EXAMPLES:

```
sage: from sage.plot.arrow import CurveArrow
sage: b = CurveArrow(path=[[(0,0),(.5,.5),(1,0)],[(.5,1),(0,0)]],options={})
sage: b
CurveArrow from (0, 0) to (0, 0)
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: from sage.plot.arrow import CurveArrow
sage: b = CurveArrow(path=[[(0,0),(.5,.5),(1,0)],[(.5,1),(0,0)]],options={})
sage: d = b.get_minmax_data()
sage: d['xmin']
0.0
sage: d['xmax']
1.0
```

sage.plot.arrow.**arrow**(*tailpoint=None*, *headpoint=None*, ***kwds*)

Returns either a 2-dimensional or 3-dimensional arrow depending on value of points.

For information regarding additional arguments, see either arrow2d? or arrow3d?.

EXAMPLES:

```
sage: arrow((0,0), (1,1))
sage: arrow((0,0,1), (1,1,1))
```

sage.plot.arrow.**arrow2d**(*tailpoint=None*, *headpoint=None*, *path=None*, *head=1*, *linestyle='solid'*, *rgbcolor=(0, 0, 1)*, *width=2*, *zorder=2*, *legend_label=None*, ***options*)

If tailpoint and headpoint are provided, returns an arrow from (xmin, ymin) to (xmax, ymax). If tailpoint or headpoint is None and path is not None, returns an arrow along the path. (See further info on paths in bezier_path).

INPUT:

- `tailpoint` - the starting point of the arrow
- `headpoint` - where the arrow is pointing to
- `path` - the list of points and control points (see bezier_path for detail) that the arrow will follow from source to destination
- `head` - 0, 1 or 2, whether to draw the head at the start (0), end (1) or both (2) of the path (using 0 will swap headpoint and tailpoint). This is ignored in 3D plotting.
- `linestyle` - (default: `'solid'`) The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.
- `width` - (default: 2) the width of the arrow shaft, in points
- `color` - (default: (0,0,1)) the color of the arrow (as an RGB tuple or a string)

- `hue` - the color of the arrow (as a number)
- `arrowsize` - the size of the arrowhead
- `arrowshorten` - the length in points to shorten the arrow (ignored if using path parameter)
- `legend_label` - the label for this item in the legend
- `legend_color` - the color for the legend label
- `zorder` - the layer level to draw the arrow– note that this is ignored in 3D plotting.

EXAMPLES:

A straight, blue arrow:

```
sage: arrow2d((1, 1), (3, 3))
```

Make a red arrow:

```
sage: arrow2d((-1, -1), (2, 3), color=(1,0,0))
sage: arrow2d((-1, -1), (2, 3), color='red')
```

You can change the width of an arrow:

```
sage: arrow2d((1, 1), (3, 3), width=5, arrowsize=15)
```

Use a dashed line instead of a solid one for the arrow:

```
sage: arrow2d((1, 1), (3, 3), linestyle='dashed')
sage: arrow2d((1, 1), (3, 3), linestyle='--')
```

A pretty circle of arrows:

```
sage: sum([arrow2d((0,0), (cos(x),sin(x)), hue=x/(2*pi)) for x in [0..2*pi,step=0.1]])
```

If we want to draw the arrow between objects, for example, the boundaries of two lines, we can use the arrow-shorten option to make the arrow shorter by a certain number of points:

```
sage: line([(0,0),(1,0)],thickness=10)+line([(0,1),(1,1)], thickness=10)+arrow2d((0.5,0),(0.5,1)
```

If BOTH headpoint and tailpoint are None, then an empty plot is returned:

```
sage: arrow2d(headpoint=None, tailpoint=None)
```

We can also draw an arrow with a legend:

```
sage: arrow((0,0), (0,2), legend_label='up', legend_color='purple')
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: arrow2d((-2, 2), (7,1), frame=True)
sage: arrow2d((-2, 2), (7,1)).show(frame=True)
```

CHAPTER

SIX

BAR CHARTS

class sage.plot.bar_chart.**BarChart**(*ind*, *datalist*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Graphics primitive that represents a bar chart.

EXAMPLES:

```
sage: from sage.plot.bar_chart import BarChart
sage: g = BarChart(range(4), [1,3,2,0], {}); g
BarChart defined by a 4 datalist
sage: type(g)
<class 'sage.plot.bar_chart.BarChart'>
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: b = bar_chart([-2.3,5,-6,12])
sage: d = b.get_minmax_data()
sage: d['xmin']
0
sage: d['xmax']
4
```

sage.plot.bar_chart.**bar_chart**(*datalist*, *aspect_ratio='automatic'*, *width=0.5*, *legend_label=None*, *rgbcolor=(0, 0, 1)*, ***options*)

A bar chart of (currently) one list of numerical data. Support for more data lists in progress.

EXAMPLES:

A bar_chart with blue bars:

```
sage: bar_chart([1,2,3,4])
```

A bar_chart with thinner bars:

```
sage: bar_chart([x^2 for x in range(1,20)], width=0.2)
```

A bar_chart with negative values and red bars:

```
sage: bar_chart([-3,5,-6,11], rgbcolor=(1,0,0))
```

A bar chart with a legend (it's possible, not necessarily useful):

```
sage: bar_chart([-1,1,-1,1], legend_label='wave')
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: bar_chart([-2,8,-7,3], rgbcolor=(1,0,0), axes=False)
sage: bar_chart([-2,8,-7,3], rgbcolor=(1,0,0)).show(axes=False) # These are equivalent
```

CHAPTER

SEVEN

BEZIER PATHS

class sage.plot.bezier_path.**BezierPath**(*path*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive_xydata

Path of Bezier Curves graphics primitive.

The input to this constructor is a list of curves, each a list of points, along which to create the curves, along with a dict of any options passed.

EXAMPLES:

```
sage: from sage.plot.bezier_path import BezierPath
sage: BezierPath([[(0,0),(.5,.5),(1,0)],[(.5,1),(0,0)]],{'linestyle':'dashed'})
Bezier path from (0, 0) to (0, 0)
```

We use bezier_path() to actually plot Bezier curves:

```
sage: bezier_path([[(0,0),(.5,.5),(1,0)],[(.5,1),(0,0)]],linestyle="dashed")
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: b = bezier_path([[(0,0),(.5,.5),(1,0)],[(.5,1),(0,0)]])
sage: d = b.get_minmax_data()
sage: d['xmin']
0.0
sage: d['xmax']
1.0
```

plot3d(*z=0*, ***kwds*)

Returns a 3D plot (Jmol) of the Bezier path. Since a BezierPath primitive contains only x, y coordinates, the path will be drawn in some plane (default is $z = 0$). To create a Bezier path with nonzero (and nonidentical) z coordinates in the path and control points, use the function bezier3d() instead of bezier_path().

EXAMPLES:

```
sage: b = bezier_path([[(0,0),(0,1),(1,0)]])
sage: A = b.plot3d()
sage: B = b.plot3d(z=2)
sage: A+B

sage: bezier3d([[(0,0,0),(1,0,0),(0,1,0),(0,1,1)]])
```

sage.plot.bezier_path.**bezier_path**(*path*, *rgbcolor=(0, 0, 0)*, *thickness=1*, *zorder=2*, *alpha=1*, *linestyle='solid'*, *fill=False*, ***options*)

Returns a Graphics object of a Bezier path corresponding to the path parameter. The path is a list of curves, and each curve is a list of points. Each point is a tuple `(x,y)`.

The first curve contains the endpoints as the first and last point in the list. All other curves assume a starting point given by the last entry in the preceding list, and take the last point in the list as their opposite endpoint. A curve can have 0, 1 or 2 control points listed between the endpoints. In the input example for path below, the first and second curves have 2 control points, the third has one, and the fourth has no control points:

path = [[p1, c1, c2, p2], [c3, c4, p3], [c5, p4], [p5], ...]

In the case of no control points, a straight line will be drawn between the two endpoints. If one control point is supplied, then the curve at each of the endpoints will be tangent to the line from that endpoint to the control point. Similarly, in the case of two control points, at each endpoint the curve will be tangent to the line connecting that endpoint with the control point immediately after or immediately preceding it in the list.

So in our example above, the curve between p1 and p2 is tangent to the line through p1 and c1 at p1, and tangent to the line through p2 and c2 at p2. Similarly, the curve between p2 and p3 is tangent to line(p2,c3) at p2 and tangent to line(p3,c4) at p3. Curve(p3,p4) is tangent to line(p3,c5) at p3 and tangent to line(p4,c5) at p4. Curve(p4,p5) is a straight line.

INPUT:

- `path` – a list of lists of tuples (see above)
- `alpha` – default: 1
- `fill` – default: False
- `thickness` – default: 1
- **`linestyle` – default: `'solid'`, The style of the line, which is one** of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.
- `rbgcolor` – default: (0,0,0)
- `zorder` – the layer in which to draw

EXAMPLES:

```
sage: path = [[(0,0),(.5,.1),(.75,3),(1,0)],[(.5,1),(.5,0)],[(.2,.5)]]
sage: b = bezier_path(path, linestyle='dashed', rgbcolor='green')
sage: b
```

To construct a simple curve, create a list containing a single list:

```
sage: path = [[(0,0),(.5,1),(1,0)]]
sage: curve = bezier_path(path, linestyle='dashed', rgbcolor='green')
sage: curve
```

Extra options will get passed on to `show()`, as long as they are valid:

```
sage: bezier_path([[(0,1),(.5,0),(1,1)]], fontsize=50)
sage: bezier_path([[(0,1),(.5,0),(1,1)]]).show(fontsize=50) # These are equivalent
```

TESTS:

We shouldn't modify our argument, trac ticket #13822:

```
sage: bp = [[(1,1),(2,3),(3,3)], [(4,4),(5,5)]]
sage: foo = bezier_path(bp)
sage: bp
[[(1, 1), (2, 3), (3, 3)], [(4, 4), (5, 5)]]
```

CHAPTER

EIGHT

CIRCLES

class sage.plot.circle.**Circle**(*x*, *y*, *r*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the Circle graphics type. See circle? for information about actually plotting circles.

INPUT:

- x - x-coordinate of center of Circle
- y - y-coordinate of center of Circle
- r - radius of Circle object
- options - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via `circle`:

```
sage: from sage.plot.circle import Circle
sage: C = Circle(2,3,5,{'zorder':2})
sage: C
Circle defined by (2.0,3.0) with r=5.0
sage: C.options()['zorder']
2
sage: C.r
5.0
```

TESTS:

We test creating a circle:

```
sage: C = circle((2,3), 5)
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: p = circle((3, 3), 1)
sage: d = p.get_minmax_data()
sage: d['xmin']
2.0
sage: d['ymin']
2.0
```

plot3d(*z=0*, ***kwds*)

Plots a 2D circle (actually a 50-gon) in 3D, with default height zero.

INPUT:

- z - optional 3D height above xy-plane.

EXAMPLES:

```
sage: circle((0,0), 1).plot3d()
```

This example uses this method implicitly, but does not pass the optional parameter z to this method:

```
sage: sum([circle((random(),random()), random()).plot3d(z=random()) for _ in range(20)])
```

These examples are explicit, and pass z to this method:

```
sage: C = circle((2,pi), 2, hue=.8, alpha=.3, fill=True)
sage: c = C[0]
sage: d = c.plot3d(z=2)
sage: d.texture.opacity
0.300000000000000

sage: C = circle((2,pi), 2, hue=.8, alpha=.3, linestyle='dotted')
sage: c = C[0]
sage: d = c.plot3d(z=2)
sage: d.jmol_repr(d.testing_render_params())[0][-1]
'color $line_1 translucent 0.7 [204,0,255]'
```

sage.plot.circle.**circle**(*center*, *radius*, *edgecolor='blue'*, *legend_color=None*, *facecolor='blue'*, *clip=True*, *linestyle='solid'*, *thickness=1*, *zorder=5*, *aspect_ratio=1.0*, *alpha=1*, *legend_label=None*, *fill=False*, ***options*)

Return a circle at a point center = (x, y) (or (x, y, z) and parallel to the xy-plane) with radius = r. Type `circle.options` to see all options.

OPTIONS:

- `alpha` - default: 1
- `fill` - default: False
- `thickness` - default: 1
- `linestyle` - default: `'solid'` (2D plotting only) The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.
- `edgecolor` - default: 'blue' (2D plotting only)
- `facecolor` - default: 'blue' (2D plotting only, useful only if `fill=True`)
- `rgbcolor` - 2D or 3D plotting. This option overrides `edgecolor` and `facecolor` for 2D plotting.
- `legend_label` - the label for this item in the legend
- `legend_color` - the color for the legend label

EXAMPLES:

The default color is blue, the default linestyle is solid, but this is easy to change:

```
sage: c = circle((1,1), 1)
sage: c

sage: c = circle((1,1), 1, rgbcolor=(1,0,0), linestyle='-.')
sage: c
```

We can also use this command to plot three-dimensional circles parallel to the xy-plane:

```
sage: c = circle((1,1,3), 1, rgbcolor=(1,0,0))
sage: c
sage: type(c)
<class 'sage.plot.plot3d.base.TransformGroup'>
```

To correct the aspect ratio of certain graphics, it is necessary to show with a `figsize` of square dimensions:

```
sage: c.show(figsize=[5,5],xmin=-1,xmax=3,ymin=-1,ymax=3)
```

Here we make a more complicated plot, with many circles of different colors:

```
sage: g = Graphics()
sage: step=6; ocur=1/5; paths=16;
sage: PI = math.pi    # numerical for speed -- fine for graphics
sage: for r in range(1,paths+1):
...       for x,y in [((r+ocur)*math.cos(n), (r+ocur)*math.sin(n)) for n in srange(0, 2*PI+PI/st
...           g += circle((x,y), ocur, rgbcolor=hue(r/paths))
...       rnext = (r+1)^2
...       ocur = (rnext-r)-ocur
...
sage: g.show(xmin=-(paths+1)^2, xmax=(paths+1)^2, ymin=-(paths+1)^2, ymax=(paths+1)^2, figsize=[
```

Note that the `rgbcolor` option overrides the other coloring options. This produces red fill in a blue circle:

```
sage: circle((2,3), 1, fill=True, edgecolor='blue')
```

This produces an all-green filled circle:

```
sage: circle((2,3), 1, fill=True, edgecolor='blue', rgbcolor='green')
```

The option `hue` overrides *all* other options, so be careful with its use. This produces a purplish filled circle:

```
sage: circle((2,3), 1, fill=True, edgecolor='blue', rgbcolor='green', hue=.8)
```

And circles with legends:

```
sage: circle((4,5), 1, rgbcolor='yellow', fill=True, legend_label='the sun').show(xmin=0, ymin=0
```

```
sage: circle((4,5), 1, legend_label='the sun', legend_color='yellow').show(xmin=0, ymin=0)
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: circle((0, 0), 2, figsize=[10,10]) # That circle is huge!
```

```
sage: circle((0, 0), 2).show(figsize=[10,10]) # These are equivalent
```

TESTS:

We cannot currently plot circles in more than three dimensions:

```
sage: circle((1,1,1,1), 1, rgbcolor=(1,0,0))
Traceback (most recent call last):
...
ValueError: The center of a plotted circle should have two or three coordinates.
```

The default aspect ratio for a circle is 1.0:

```
sage: P = circle((1,1), 1)
sage: P.aspect_ratio()
1.0
```

CHAPTER

NINE

COMPLEX PLOTS

Complex Plots

class sage.plot.complex_plot.**ComplexPlot**(*rgb_data*, *xrange*, *yrange*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

The GraphicsPrimitive to display complex functions in using the domain coloring method

INPUT:

- rgb_data – An array of colored points to be plotted.
- xrange – A minimum and maximum x value for the plot.
- yrange – A minimum and maximum y value for the plot.

TESTS:

```
sage: p = complex_plot(lambda z: z^2-1, (-2, 2), (-2, 2))
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: p = complex_plot(lambda z: z, (-1, 2), (-3, 4))
sage: sorted(p.get_minmax_data().items())
[('xmax', 2.0), ('xmin', -1.0), ('ymax', 4.0), ('ymin', -3.0)]
```

sage.plot.complex_plot.**complex_plot**(*f*, *xrange*, *yrange*, *plot_points=100*, *interpolation='catrom'*, ***options*)

complex_plot takes a complex function of one variable, $f(z)$ and plots output of the function over the specified xrange and yrange as demonstrated below. The magnitude of the output is indicated by the brightness (with zero being black and infinity being white) while the argument is represented by the hue (with red being positive real, and increasing through orange, yellow, ... as the argument increases).

```
complex_plot(f, (xmin, xmax), (ymin, ymax), ...)
```

INPUT:

- f – a function of a single complex value $x + iy$
- (xmin, xmax) – 2-tuple, the range of x values
- (ymin, ymax) – 2-tuple, the range of y values

The following inputs must all be passed in as named parameters:

- plot_points – integer (default: 100); number of points to plot in each direction of the grid

- interpolation – string (default: 'catrom'), the interpolation method to use: 'bilinear', 'bicubic', 'spline16', 'spline36', 'quadric', 'gaussian', 'sinc', 'bessel', 'mitchell', 'lanczos', 'catrom', 'hermite', 'hanning', 'hamming', 'kaiser'

EXAMPLES:

Here we plot a couple of simple functions:

```
sage: complex_plot(sqrt(x), (-5, 5), (-5, 5))

sage: complex_plot(sin(x), (-5, 5), (-5, 5))

sage: complex_plot(log(x), (-10, 10), (-10, 10))

sage: complex_plot(exp(x), (-10, 10), (-10, 10))
```

A function with some nice zeros and a pole:

```
sage: f(z) = z^5 + z - 1 + 1/z
sage: complex_plot(f, (-3, 3), (-3, 3))
```

Here is the identity, useful for seeing what values map to what colors:

```
sage: complex_plot(lambda z: z, (-3, 3), (-3, 3))
```

The Riemann Zeta function:

```
sage: complex_plot(zeta, (-30,30), (-30,30))
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: complex_plot(lambda z: z, (-3, 3), (-3, 3), figsize=[1,1])

sage: complex_plot(lambda z: z, (-3, 3), (-3, 3)).show(figsize=[1,1]) # These are equivalent
```

TESTS:

Test to make sure that using fast_callable functions works:

```
sage: f(x) = x^2
sage: g = fast_callable(f, domain=CC, vars='x')
sage: h = fast_callable(f, domain=CDF, vars='x')
sage: P = complex_plot(f, (-10, 10), (-10, 10))
sage: Q = complex_plot(g, (-10, 10), (-10, 10))
sage: R = complex_plot(h, (-10, 10), (-10, 10))
sage: S = complex_plot(exp(x)-sin(x), (-10, 10), (-10, 10))
sage: P; Q; R; S
```

Test to make sure symbolic functions still work without declaring a variable. (We don't do this in practice because it doesn't use fast_callable, so it is much slower.)

```
sage: complex_plot(sqrt, (-5, 5), (-5, 5))
```

sage.plot.complex_plot.**complex_to_rgb**(*z_values*)

INPUT:

- z_values – A grid of complex numbers, as a list of lists

OUTPUT:

An $N \times M \times 3$ floating point Numpy array X, where X[i, j] is an (r,g,b) tuple.

EXAMPLES:

```
sage: from sage.plot.complex_plot import complex_to_rgb
sage: complex_to_rgb([[0, 1, 1000]])
array([[[ 0.        ,  0.        ,  0.        ],
        [ 0.77172568,  0.        ,  0.        ],
        [ 1.        ,  0.64421177,  0.64421177]]])
sage: complex_to_rgb([[0, 1j, 1000j]])
array([[[ 0.        ,  0.        ,  0.        ],
        [ 0.38586284,  0.77172568,  0.        ],
        [ 0.82210588,  1.        ,  0.64421177]]])
```

CHAPTER

TEN

CONTOUR PLOTS

class sage.plot.contour_plot.**ContourPlot**(*xy_data_array*, *xrange*, *yrange*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the contour plot graphics type. See `contour_plot?` for help actually doing contour plots.

INPUT:

- `xy_data_array` - list of lists giving evaluated values of the function on the grid
- `xrange` - tuple of 2 floats indicating range for horizontal direction
- `yrange` - tuple of 2 floats indicating range for vertical direction
- `options` - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via `contour_plot`:

```
sage: from sage.plot.contour_plot import ContourPlot
sage: C = ContourPlot([[1,3],[2,4]],(1,2),(2,3),options={})
sage: C
ContourPlot defined by a 2 x 2 data grid
sage: C.xrange
(1, 2)
```

TESTS:

We test creating a contour plot:

```
sage: x,y = var('x,y')
sage: contour_plot(x^2-y^3+10*sin(x*y), (x, -4, 4), (y, -4, 4),plot_points=121,cmap='hsv')
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f(x,y) = x^2 + y^2
sage: d = contour_plot(f, (3, 6), (3, 6))[0].get_minmax_data()
sage: d['xmin']
3.0
sage: d['ymin']
3.0
```

sage.plot.contour_plot.**contour_plot**(*f*, *xrange*, *yrange*, *axes=False*, *linestyles=None*, *region=None*, *labels=False*, *plot_points=100*, *linewidths=None*, *colorbar=False*, *contours=None*, *aspect_ratio=1*, *legend_label=None*, *frame=True*, *fill=True*, *label_inline=None*, *label_fmt='%1.2f'*, *label_fontsize=9*, *label_colors='blue'*, *label_inline_spacing=3*, *colorbar_spacing=None*, *colorbar_orientation='vertical'*, *colorbar_format=None*, ***options*)

`contour_plot` takes a function of two variables, $f(x, y)$ and plots contour lines of the function over the specified `xrange` and `yrange` as demonstrated below.

```
contour_plot(f, (xmin, xmax), (ymin, ymax), ...)
```

INPUT:

- `f` – a function of two variables
- `(xmin, xmax)` – 2-tuple, the range of `x` values OR 3-tuple `(x,xmin,xmax)`
- `(ymin, ymax)` – 2-tuple, the range of `y` values OR 3-tuple `(y,ymin,ymax)`

The following inputs must all be passed in as named parameters:

- `plot_points` – integer (default: 100); number of points to plot in each direction of the grid. For old computers, 25 is fine, but should not be used to verify specific intersection points.
- `fill` – bool (default: `True`), whether to color in the area between contour lines
- `cmap` – a colormap (default: `'gray'`), the name of a predefined colormap, a list of colors or an instance of a matplotlib Colormap. Type: `import matplotlib.cm; matplotlib.cm.datad.keys()` for available colormap names.
- `contours` – integer or list of numbers (default: `None`): If a list of numbers is given, then this specifies the contour levels to use. If an integer is given, then this many contour lines are used, but the exact levels are determined automatically. If `None` is passed (or the option is not given), then the number of contour lines is determined automatically, and is usually about 5.
- `linewidths` – integer or list of integer (default: None), if a single integer all levels will be of the width given, otherwise the levels will be plotted with the width in the order given. If the list is shorter than the number of contours, then the widths will be repeated cyclically.
- `linestyles` – string or list of strings (default: None), the style of the lines to be plotted, one of: `"solid"`, `"dashed"`, `"dashdot"`, `"dotted"`, respectively `"-"`, `"--"`, `"-."`, `":"`. If the list is shorter than the number of contours, then the styles will be repeated cyclically.
- `labels` – boolean (default: False) Show level labels or not.

  The following options are to adjust the style and placement of labels, they have no effect if no labels are shown.

  - `label_fontsize` – integer (default: 9), the font size of the labels.
  - `label_colors` – string or sequence of colors (default: None) If a string, gives the name of a single color with which to draw all labels. If a sequence, gives the colors of the labels. A color is a string giving the name of one or a 3-tuple of floats.
  - `label_inline` – boolean (default: False if fill is True, otherwise True), controls whether the underlying contour is removed or not.
  - `label_inline_spacing` – integer (default: 3), When inline, this is the amount of contour that is removed from each side, in pixels.

- `label_fmt` – a format string (default: "%1.2f"), this is used to get the label text from the level. This can also be a dictionary with the contour levels as keys and corresponding text string labels as values. It can also be any callable which returns a string when called with a numeric contour level.

- `colorbar` – boolean (default: False) Show a colorbar or not.

  The following options are to adjust the style and placement of colorbars. They have no effect if a colorbar is not shown.

  - `colorbar_orientation` – string (default: 'vertical'), controls placement of the colorbar, can be either 'vertical' or 'horizontal'

  - `colorbar_format` – a format string, this is used to format the colorbar labels.

  - `colorbar_spacing` – string (default: 'proportional'). If 'proportional', make the contour divisions proportional to values. If 'uniform', space the colorbar divisions uniformly, without regard for numeric values.

- `legend_label` – the label for this item in the legend

- **`region` - (default: None) If region is given, it must be a function** of two variables. Only segments of the surface where region(x,y) returns a number >0 will be included in the plot.

EXAMPLES:

Here we plot a simple function of two variables. Note that since the input function is an expression, we need to explicitly declare the variables in 3-tuples for the range:

```
sage: x,y = var('x,y')
sage: contour_plot(cos(x^2+y^2), (x, -4, 4), (y, -4, 4))
```

Here we change the ranges and add some options:

```
sage: x,y = var('x,y')
sage: contour_plot((x^2)*cos(x*y), (x, -10, 5), (y, -5, 5), fill=False, plot_points=150)
```

An even more complicated plot:

```
sage: x,y = var('x,y')
sage: contour_plot(sin(x^2 + y^2)*cos(x)*sin(y), (x, -4, 4), (y, -4, 4),plot_points=150)
```

Some elliptic curves, but with symbolic endpoints. In the first example, the plot is rotated 90 degrees because we switch the variables x, y:

```
sage: x,y = var('x,y')
sage: contour_plot(y^2 + 1 - x^3 - x, (y,-pi,pi), (x,-pi,pi))

sage: contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi))
```

We can play with the contour levels:

```
sage: x,y = var('x,y')
sage: f(x,y) = x^2 + y^2
sage: contour_plot(f, (-2, 2), (-2, 2))

sage: contour_plot(f, (-2, 2), (-2, 2), contours=2, cmap=[(1,0,0), (0,1,0), (0,0,1)])

sage: contour_plot(f, (-2, 2), (-2, 2), contours=(0.1, 1.0, 1.2, 1.4), cmap='hsv')

sage: contour_plot(f, (-2, 2), (-2, 2), contours=(1.0,), fill=False)
```

```
sage: contour_plot(x-y^2,(x,-5,5),(y,-3,3),contours=[-4,0,1])
```

We can change the style of the lines:

```
sage: contour_plot(f, (-2,2), (-2,2), fill=False, linewidths=10)

sage: contour_plot(f, (-2,2), (-2,2), fill=False, linestyles='dashdot')

sage: P=contour_plot(x^2-y^2,(x,-3,3),(y,-3,3),contours=[0,1,2,3,4],\
...    linewidths=[1,5],linestyles=['solid','dashed'],fill=False)
sage: P

sage: P=contour_plot(x^2-y^2,(x,-3,3),(y,-3,3),contours=[0,1,2,3,4],\
...    linewidths=[1,5],linestyles=['solid','dashed'])
sage: P

sage: P=contour_plot(x^2-y^2,(x,-3,3),(y,-3,3),contours=[0,1,2,3,4],\
...    linewidths=[1,5],linestyles=['-',':'])
sage: P
```

We can add labels and play with them:

```
sage: contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),  fill=False, cmap='hsv', labels=Tr

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), fill=False, cmap='hsv',\
...     labels=True, label_fmt="%1.0f", label_colors='black')
sage: P

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), fill=False, cmap='hsv',labels=Tr
...    contours=[-4,0,4],  label_fmt={-4:"low", 0:"medium", 4: "hi"}, label_colors='black')
sage: P

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), fill=False, cmap='hsv',labels=Tr
...    contours=[-4,0,4],  label_fmt=lambda x: "$z=%s$"%x, label_colors='black', label_inline=Tr
...    label_fontsize=12)
sage: P

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), \
...    fill=False, cmap='hsv', labels=True, label_fontsize=18)
sage: P

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), \
...    fill=False, cmap='hsv', labels=True, label_inline_spacing=1)
sage: P

sage: P= contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi), \
...    fill=False, cmap='hsv', labels=True, label_inline=False)
sage: P
```

We can change the color of the labels if so desired:

```
sage: contour_plot(f, (-2,2), (-2,2), labels=True, label_colors='red')
```

We can add a colorbar as well:

```
sage: f(x,y)=x^2-y^2
sage: contour_plot(f, (x,-3,3), (y,-3,3), colorbar=True)
```

```
sage: contour_plot(f, (x,-3,3), (y,-3,3), colorbar=True,colorbar_orientation='horizontal')

sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[-2,-1,4],colorbar=True)

sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[-2,-1,4],colorbar=True,colorbar_spacing='uni

sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[0,2,3,6],colorbar=True,colorbar_format='%.3f

sage: contour_plot(f, (x,-3,3), (y,-3,3), labels=True,label_colors='red',contours=[0,2,3,6],colo

sage: contour_plot(f, (x,-3,3), (y,-3,3), cmap='winter', contours=20, fill=False, colorbar=True)
```

This should plot concentric circles centered at the origin:

```
sage: x,y = var('x,y')
sage: contour_plot(x^2+y^2-2,(x,-1,1), (y,-1,1))
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: f(x, y) = cos(x) + sin(y)
sage: contour_plot(f, (0, pi), (0, pi), axes=True)
```

One can also plot over a reduced region:

```
sage: contour_plot(x**2-y**2, (x,-2, 2), (y,-2, 2),region=x-y,plot_points=300)

sage: contour_plot(f, (0, pi), (0, pi)).show(axes=True) # These are equivalent
```

Note that with `fill=False` and grayscale contours, there is the possibility of confusion between the contours and the axes, so use `fill=False` together with `axes=True` with caution:

```
sage: contour_plot(f, (-pi, pi), (-pi, pi), fill=False, axes=True)
```

TESTS:

To check that ticket 5221 is fixed, note that this has three curves, not two:

```
sage: x,y = var('x,y')
sage: contour_plot(x-y^2,(x,-5,5),(y,-3,3),contours=[-4,-2,0], fill=False)
```

sage.plot.contour_plot.**equify**(*f*)

Returns the equation rewritten as a symbolic function to give negative values when True, positive when False.

EXAMPLES:

```
sage: from sage.plot.contour_plot import equify
sage: var('x, y')
(x, y)
sage: equify(x^2 < 2)
x^2 - 2
sage: equify(x^2 > 2)
-x^2 + 2
sage: equify(x*y > 1)
-x*y + 1
sage: equify(y > 0)
-y
sage: f=equify(lambda x,y: x>y)
sage: f(1,2)
1
sage: f(2,1)
-1
```

sage.plot.contour_plot.**implicit_plot**(*f, xrange, yrange, cmap=['blue'], contours=(0, 0), plot_points=150, fill=False, **options*)

implicit_plot takes a function of two variables, $f(x, y)$ and plots the curve $f(x, y) = 0$ over the specified xrange and yrange as demonstrated below.

```
implicit_plot(f, (xmin, xmax), (ymin, ymax), ...)
```

```
implicit_plot(f, (x, xmin, xmax), (y, ymin, ymax), ...)
```

INPUT:

- f – a function of two variables or equation in two variables
- (xmin, xmax) – 2-tuple, the range of x values or (x,xmin,xmax)
- (ymin, ymax) – 2-tuple, the range of y values or (y,ymin,ymax)

The following inputs must all be passed in as named parameters:

- plot_points – integer (default: 150); number of points to plot in each direction of the grid
- fill – boolean (default: False); if True, fill the region $f(x, y) < 0$.
- linewidth – integer (default: None), if a single integer all levels will be of the width given, otherwise the levels will be plotted with the widths in the order given.
- linestyle – string (default: None), the style of the line to be plotted, one of: "solid", "dashed", "dashdot" or "dotted", respectively "-", "--", "-.", or ":".
- color – string (default: blue), the color of the plot. Colors are defined in sage.plot.colors; try colors? to see them all.
- legend_label – the label for this item in the legend
- base - (default: 10) the base of the logarithm if a logarithmic scale is set. This must be greater than 1. The base can be also given as a list or tuple (basex, basey). basex sets the base of the logarithm along the horizontal axis and basey sets the base along the vertical axis.
- scale – (default: "linear") string. The scale of the axes. Possible values are "linear", "loglog", "semilogx", "semilogy".

  The scale can be also be given as single argument that is a list or tuple (scale, base) or (scale, basex, basey).

  The "loglog" scale sets both the horizontal and vertical axes to logarithmic scale. The "semilogx" scale sets the horizontal axis to logarithmic scale. The "semilogy" scale sets the vertical axis to logarithmic scale. The "linear" scale is the default value when Graphics is initialized.

EXAMPLES:

A simple circle with a radius of 2. Note that since the input function is an expression, we need to explicitly declare the variables in 3-tuples for the range:

```
sage: var("x y")
(x, y)
sage: implicit_plot(x^2+y^2-2, (x,-3,3), (y,-3,3))
```

I can do the same thing, but using a callable function so I don't need to explicitly define the variables in the ranges, and filling the inside:

```
sage: f(x,y) = x^2 + y^2 - 2
sage: implicit_plot(f, (-3, 3), (-3, 3),fill=True)
```

The same circle but with a different line width:

```
sage: implicit_plot(f, (-3,3), (-3,3), linewidth=6)
```

And again the same circle but this time with a dashdot border:

```
sage: implicit_plot(f, (-3,3), (-3,3), linestyle='dashdot')
```

You can also plot an equation:

```
sage: var("x y")
(x, y)
sage: implicit_plot(x^2+y^2 == 2, (x,-3,3), (y,-3,3))
```

You can even change the color of the plot:

```
sage: implicit_plot(x^2+y^2 == 2, (x,-3,3), (y,-3,3), color="red")
```

Here is a beautiful (and long) example which also tests that all colors work with this:

```
sage: G = Graphics()
sage: counter = 0
sage: for col in colors.keys(): # long time
...       G += implicit_plot(x^2+y^2==1+counter*.1, (x,-4,4),(y,-4,4),color=col)
...       counter += 1
sage: G.show(frame=False)
```

We can define a level-n approximation of the boundary of the Mandelbrot set:

```
sage: def mandel(n):
...       c = polygen(CDF, 'c')
...       z = 0
...       for i in range(n):
...           z = z*z + c
...       def f(x, y):
...           val = z(CDF(x, y))
...           return val.norm() - 4
...       return f
```

The first-level approximation is just a circle:

```
sage: implicit_plot(mandel(1), (-3, 3), (-3, 3))
```

A third-level approximation starts to get interesting:

```
sage: implicit_plot(mandel(3), (-2, 1), (-1.5, 1.5))
```

The seventh-level approximation is a degree 64 polynomial, and `implicit_plot` does a pretty good job on this part of the curve. (`plot_points=200` looks even better, but it takes over a second.)

```
sage: implicit_plot(mandel(7), (-0.3, 0.05), (-1.15, -0.9),plot_points=50)
```

When making a filled implicit plot using a python function rather than a symbolic expression the user should increase the number of plot points to avoid artifacts:

```
sage: implicit_plot(lambda x,y: x^2+y^2-2, (x,-3,3), (y,-3,3), fill=True, plot_points=500) # lon
```

An example of an implicit plot on 'loglog' scale:

```
sage: implicit_plot(x^2+y^2 == 200, (x,1,200), (y,1,200), scale='loglog')
```

TESTS:

```
sage: f(x,y) = x^2 + y^2 - 2
sage: implicit_plot(f, (-3, 3), (-3, 3),fill=5)
Traceback (most recent call last):
...
ValueError: fill=5 is not supported
```

sage.plot.contour_plot.**region_plot**(*f*, *xrange*, *yrange*, *plot_points*, *incol*, *outcol*, *bordercol*, *borderstyle*, *borderwidth*, *frame=False*, *axes=True*, *plot_points=100*, *aspect_ratio=1*, *outcol='white'*, *borderwidth=None*, *incol='blue'*, *borderstyle=None*, *legend_label=None*, *bordercol=None*, ***options*)

`region_plot` takes a boolean function of two variables, $f(x, y)$ and plots the region where f is True over the specified `xrange` and `yrange` as demonstrated below.

`region_plot(f, (xmin, xmax), (ymin, ymax), ...)`

INPUT:

- `f` – a boolean function of two variables
- `(xmin, xmax)` – 2-tuple, the range of `x` values OR 3-tuple `(x,xmin,xmax)`
- `(ymin, ymax)` – 2-tuple, the range of `y` values OR 3-tuple `(y,ymin,ymax)`
- `plot_points` – integer (default: 100); number of points to plot in each direction of the grid
- `incol` – a color (default: `'blue'`), the color inside the region
- `outcol` – a color (default: `'white'`), the color of the outside of the region

If any of these options are specified, the border will be shown as indicated, otherwise it is only implicit (with color `incol`) as the border of the inside of the region.

- `bordercol` – a color (default: `None`), the color of the border (`'black'` if `borderwidth` or `borderstyle` is specified but not `bordercol`)
- `borderstyle` – string (default: 'solid'), one of `'solid'`, `'dashed'`, `'dotted'`, `'dashdot'`, respectively `'-'`, `'--'`, `':'`, `'-.'`.
- `borderwidth` – integer (default: None), the width of the border in pixels
- `legend_label` – the label for this item in the legend
- `base` - (default: 10) the base of the logarithm if a logarithmic scale is set. This must be greater than 1. The base can be also given as a list or tuple `(basex, basey)`. `basex` sets the base of the logarithm along the horizontal axis and `basey` sets the base along the vertical axis.
- `scale` – (default: `"linear"`) string. The scale of the axes. Possible values are `"linear"`, `"loglog"`, `"semilogx"`, `"semilogy"`.

  The scale can be also be given as single argument that is a list or tuple `(scale, base)` or `(scale, basex, basey)`.

  The `"loglog"` scale sets both the horizontal and vertical axes to logarithmic scale. The `"semilogx"` scale sets the horizontal axis to logarithmic scale. The `"semilogy"` scale sets the vertical axis to logarithmic scale. The `"linear"` scale is the default value when `Graphics` is initialized.

EXAMPLES:

Here we plot a simple function of two variables:

```
sage: x,y = var('x,y')
sage: region_plot(cos(x^2+y^2) <= 0, (x, -3, 3), (y, -3, 3))
```

Here we play with the colors:

```
sage: region_plot(x^2+y^3 < 2, (x, -2, 2), (y, -2, 2), incol='lightblue', bordercol='gray')
```

An even more complicated plot, with dashed borders:

```
sage: region_plot(sin(x)*sin(y) >= 1/4, (x,-10,10), (y,-10,10), incol='yellow', bordercol='black
```

A disk centered at the origin:

```
sage: region_plot(x^2+y^2<1, (x,-1,1), (y,-1,1))
```

A plot with more than one condition (all conditions must be true for the statement to be true):

```
sage: region_plot([x^2+y^2<1, x<y], (x,-2,2), (y,-2,2))
```

Since it doesn't look very good, let's increase `plot_points`:

```
sage: region_plot([x^2+y^2<1, x<y], (x,-2,2), (y,-2,2), plot_points=400)
```

To get plots where only one condition needs to be true, use a function. Using lambda functions, we definitely need the extra `plot_points`:

```
sage: region_plot(lambda x,y: x^2+y^2<1 or x<y, (x,-2,2), (y,-2,2), plot_points=400)
```

The first quadrant of the unit circle:

```
sage: region_plot([y>0, x>0, x^2+y^2<1], (x,-1.1, 1.1), (y,-1.1, 1.1), plot_points = 400)
```

Here is another plot, with a huge border:

```
sage: region_plot(x*(x-1)*(x+1)+y^2<0, (x, -3, 2), (y, -3, 3), incol='lightblue', bordercol='gra
```

If we want to keep only the region where x is positive:

```
sage: region_plot([x*(x-1)*(x+1)+y^2<0, x>-1], (x, -3, 2), (y, -3, 3), incol='lightblue', plot_p
```

Here we have a cut circle:

```
sage: region_plot([x^2+y^2<4, x>-1], (x, -2, 2), (y, -2, 2), incol='lightblue', bordercol='gray'
```

The first variable range corresponds to the horizontal axis and the second variable range corresponds to the vertical axis:

```
sage: s,t=var('s,t')
sage: region_plot(s>0,(t,-2,2),(s,-2,2))

sage: region_plot(s>0,(s,-2,2),(t,-2,2))
```

An example of a region plot in 'loglog' scale:

```
sage: region_plot(x^2+y^2<100, (x,1,10), (y,1,10), scale='loglog')
```

CHAPTER

ELEVEN

DENSITY PLOTS

class sage.plot.density_plot.**DensityPlot**(*xy_data_array*, *xrange*, *yrange*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the density plot graphics type. See `density_plot?` for help actually doing density plots.

INPUT:

- `xy_data_array` - list of lists giving evaluated values of the function on the grid
- `xrange` - tuple of 2 floats indicating range for horizontal direction
- `yrange` - tuple of 2 floats indicating range for vertical direction
- `options` - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via *density_plot*:

```
sage: from sage.plot.density_plot import DensityPlot
sage: D = DensityPlot([[1,3],[2,4]],(1,2),(2,3),options={})
sage: D
DensityPlot defined by a 2 x 2 data grid
sage: D.yrange
(2, 3)
sage: D.options()
{}
```

TESTS:

We test creating a density plot:

```
sage: x,y = var('x,y')
sage: density_plot(x^2-y^3+10*sin(x*y), (x, -4, 4), (y, -4, 4),plot_points=121,cmap='hsv')
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f(x, y) = x^2 + y^2
sage: d = density_plot(f, (3, 6), (3, 6))[0].get_minmax_data()
sage: d['xmin']
3.0
sage: d['ymin']
3.0
```

sage.plot.density_plot.**density_plot**(*f*, *xrange*, *yrange*, *cmap='gray'*, *plot_points=25*, *interpolation='catrom'*, ***options*)

density_plot takes a function of two variables, $f(x, y)$ and plots the height of of the function over the specified xrange and yrange as demonstrated below.

density_plot(f, (xmin, xmax), (ymin, ymax), ...)

INPUT:

- f – a function of two variables
- (xmin, xmax) – 2-tuple, the range of x values OR 3-tuple (x,xmin,xmax)
- (ymin, ymax) – 2-tuple, the range of y values OR 3-tuple (y,ymin,ymax)

The following inputs must all be passed in as named parameters:

- plot_points – integer (default: 25); number of points to plot in each direction of the grid
- cmap – a colormap (type cmap_help() for more information).
- interpolation – string (default: 'catrom'), the interpolation method to use: 'bilinear', 'bicubic', 'spline16', 'spline36', 'quadric', 'gaussian', 'sinc', 'bessel', 'mitchell','lanczos','catrom','hermite','hanning','hamming','kaiser'

EXAMPLES:

Here we plot a simple function of two variables. Note that since the input function is an expression, we need to explicitly declare the variables in 3-tuples for the range:

```
sage: x,y = var('x,y')
sage: density_plot(sin(x)*sin(y), (x, -2, 2), (y, -2, 2))
```

Here we change the ranges and add some options; note that here f is callable (has variables declared), so we can use 2-tuple ranges:

```
sage: x,y = var('x,y')
sage: f(x,y) = x^2*cos(x*y)
sage: density_plot(f, (x,-10,5), (y, -5,5), interpolation='sinc', plot_points=100)
```

An even more complicated plot:

```
sage: x,y = var('x,y')
sage: density_plot(sin(x^2 + y^2)*cos(x)*sin(y), (x, -4, 4), (y, -4, 4), cmap='jet', plot_points
```

This should show a "spotlight" right on the origin:

```
sage: x,y = var('x,y')
sage: density_plot(1/(x^10+y^10), (x, -10, 10), (y, -10, 10))
```

Some elliptic curves, but with symbolic endpoints. In the first example, the plot is rotated 90 degrees because we switch the variables x, y:

```
sage: density_plot(y^2 + 1 - x^3 - x, (y,-pi,pi), (x,-pi,pi))

sage: density_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi))
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: density_plot(log(x) + log(y), (x, 1, 10), (y, 1, 10), dpi=20)

sage: density_plot(log(x) + log(y), (x, 1, 10), (y, 1, 10)).show(dpi=20) # These are equivalent
```

CHAPTER

TWELVE

DISKS

class sage.plot.disk.**Disk**(*point*, *r*, *angle*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the Disk graphics type. See disk? for information about actually plotting a disk (the Sage term for a sector or wedge of a circle).

INPUT:

- point - coordinates of center of disk
- r - radius of disk
- angle - beginning and ending angles of disk (i.e. angle extent of sector/wedge)
- options - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via disk:

```
sage: from sage.plot.disk import Disk
sage: D = Disk((1,2), 2, (pi/2,pi), {'zorder':3})
sage: D
Disk defined by (1.0,2.0) with r=2.0 spanning (1.57079632679, 3.14159265359) radians
sage: D.options()['zorder']
3
sage: D.x
1.0
```

TESTS:

We test creating a disk:

```
sage: disk((2,3), 2, (0,pi/2))
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: D = disk((5,4), 1, (pi/2, pi))
sage: d = D.get_minmax_data()
sage: d['xmin']
4.0
sage: d['ymin']
3.0
sage: d['xmax']
6.0
```

```
sage: d['ymax']
5.0
```

plot3d(*z=0*, ***kwds*)

Plots a 2D disk (actually a 52-gon) in 3D, with default height zero.

INPUT:

- z - optional 3D height above xy-plane.

AUTHORS:

- Karl-Dieter Crisman (05-09)

EXAMPLES:

```
sage: disk((0,0), 1, (0, pi/2)).plot3d()
sage: disk((0,0), 1, (0, pi/2)).plot3d(z=2)
sage: disk((0,0), 1, (pi/2, 0), fill=False).plot3d(3)
```

These examples show that the appropriate options are passed:

```
sage: D = disk((2,3), 1, (pi/4,pi/3), hue=.8, alpha=.3, fill=True)
sage: d = D[0]
sage: d.plot3d(z=2).texture.opacity
0.300000000000000
```

```
sage: D = disk((2,3), 1, (pi/4,pi/3), hue=.8, alpha=.3, fill=False)
sage: d = D[0]
sage: dd = d.plot3d(z=2)
sage: dd.jmol_repr(dd.testing_render_params())[0][-1]
'color $line_4 translucent 0.7 [204,0,255]'
```

sage.plot.disk.**disk**(*point*, *radius*, *angle*, *rgbcolor=(0, 0, 1)*, *thickness=0*, *aspect_ratio=1.0*, *alpha=1*, *legend_label=None*, *fill=True*, ***options*)

A disk (that is, a sector or wedge of a circle) with center at a point = (x, y) (or (x, y, z) and parallel to the xy-plane) with radius = r spanning (in radians) angle='(rad1, rad2)'.

Type `disk.options` to see all options.

EXAMPLES:

Make some dangerous disks:

```
sage: bl = disk((0.0,0.0), 1, (pi, 3*pi/2), color='yellow')
sage: tr = disk((0.0,0.0), 1, (0, pi/2), color='yellow')
sage: tl = disk((0.0,0.0), 1, (pi/2, pi), color='black')
sage: br = disk((0.0,0.0), 1, (3*pi/2, 2*pi), color='black')
sage: P  = tl+tr+bl+br
sage: P.show(xmin=-2,xmax=2,ymin=-2,ymax=2)
```

The default aspect ratio is 1.0:

```
sage: disk((0.0,0.0), 1, (pi, 3*pi/2)).aspect_ratio()
1.0
```

Another example of a disk:

```
sage: bl = disk((0.0,0.0), 1, (pi, 3*pi/2), rgbcolor=(1,1,0))
sage: bl.show(figsize=[5,5])
```

Note that since `thickness` defaults to zero, it is best to change that option when using `fill=False`:

```
sage: disk((2,3), 1, (pi/4,pi/3), hue=.8, alpha=.3, fill=False, thickness=2)
```

The previous two examples also illustrate using `hue` and `rgbcolor` as ways of specifying the color of the graphic.

We can also use this command to plot three-dimensional disks parallel to the xy-plane:

```
sage: d = disk((1,1,3), 1, (pi,3*pi/2), rgbcolor=(1,0,0))
sage: d
sage: type(d)
<type 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

Extra options will get passed on to `show()`, as long as they are valid:

```
sage: disk((0, 0), 5, (0, pi/2), xmin=0, xmax=5, ymin=0, ymax=5, figsize=(2,2), rgbcolor=(1, 0,
sage: disk((0, 0), 5, (0, pi/2), rgbcolor=(1, 0, 1)).show(xmin=0, xmax=5, ymin=0, ymax=5, figsiz
```

TESTS:

Testing that legend labels work right:

```
sage: disk((2,4), 3, (pi/8, pi/4), hue=1, legend_label='disk', legend_color='blue')
```

We cannot currently plot disks in more than three dimensions:

```
sage: d = disk((1,1,1,1), 1, (0,pi))
Traceback (most recent call last):
...
ValueError: The center point of a plotted disk should have two or three coordinates.
```

CHAPTER

THIRTEEN

ELLIPSES

class sage.plot.ellipse.**Ellipse**(*x*, *y*, *r1*, *r2*, *angle*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the Ellipse graphics type. See ellipse? for information about actually plotting ellipses.

INPUT:

- x, y - coordinates of the center of the ellipse
- r1, r2 - radii of the ellipse
- angle - angle
- options - dictionary of options

EXAMPLES:

Note that this construction should be done using ellipse:

```
sage: from sage.plot.ellipse import Ellipse
sage: Ellipse(0, 0, 2, 1, pi/4, {})
Ellipse centered at (0.0, 0.0) with radii (2.0, 1.0) and angle 0.785398163397
```

get_minmax_data()

Returns a dictionary with the bounding box data.

The bounding box is computed to be as minimal as possible.

EXAMPLES:

An example without an angle:

```
sage: p = ellipse((-2, 3), 1, 2)
sage: d = p.get_minmax_data()
sage: d['xmin']
-3.0
sage: d['xmax']
-1.0
sage: d['ymin']
1.0
sage: d['ymax']
5.0
```

The same example with a rotation of angle $\pi/2$:

```
sage: p = ellipse((-2, 3), 1, 2, pi/2)
sage: d = p.get_minmax_data()
sage: d['xmin']
-4.0
```

```
sage: d['xmax']
0.0
sage: d['ymin']
2.0
sage: d['ymax']
4.0
```

plot3d()

Plotting in 3D is not implemented.

TESTS:

```
sage: from sage.plot.ellipse import Ellipse
sage: Ellipse(0,0,2,1,pi/4,{}).plot3d()
Traceback (most recent call last):
...
NotImplementedError
```

sage.plot.ellipse.**ellipse**(*center*, *r1*, *r2*, *angle=0*, *edgecolor='blue'*, *legend_color=None*, *facecolor='blue'*, *linestyle='solid'*, *thickness=1*, *zorder=5*, *aspect_ratio=1.0*, *alpha=1*, *legend_label=None*, *fill=False*, ***options*)

Return an ellipse centered at a point center = `(x,y)` with radii = `r1,r2` and angle `angle`. Type `ellipse.options` to see all options.

INPUT:

- `center` - 2-tuple of real numbers - coordinates of the center
- `r1`, `r2` - positive real numbers - the radii of the ellipse
- `angle` - real number (default: 0) - the angle between the first axis and the horizontal

OPTIONS:

- `alpha` - default: 1 - transparency
- `fill` - default: False - whether to fill the ellipse or not
- `thickness` - default: 1 - thickness of the line
- `linestyle` - default: `'solid'` - The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.
- `edgecolor` - default: 'black' - color of the contour
- `facecolor` - default: 'red' - color of the filling
- `rgbcolor` - 2D or 3D plotting. This option overrides `edgecolor` and `facecolor` for 2D plotting.
- `legend_label` - the label for this item in the legend
- `legend_color` - the color for the legend label

EXAMPLES:

An ellipse centered at (0,0) with major and minor axes of lengths 2 and 1. Note that the default color is blue:

```
sage: ellipse((0,0),2,1)
```

More complicated examples with tilted axes and drawing options:

```
sage: ellipse((0,0),3,1,pi/6,fill=True,alpha=0.3,linestyle="dashed")
sage: ellipse((0,0),3,1,pi/6,fill=True,alpha=0.3,linestyle="--")
```

```
sage: ellipse((0,0),3,1,pi/6,fill=True,edgecolor='black',facecolor='red')
```

We see that `rgbcolor` overrides these other options, as this plot is green:

```
sage: ellipse((0,0),3,1,pi/6,fill=True,edgecolor='black',facecolor='red',rgbcolor='green')
```

The default aspect ratio for ellipses is 1.0:

```
sage: ellipse((0,0),2,1).aspect_ratio()
1.0
```

One cannot yet plot ellipses in 3D:

```
sage: ellipse((0,0,0),2,1)
Traceback (most recent call last):
...
NotImplementedError: plotting ellipse in 3D is not implemented
```

We can also give ellipses a legend:

```
sage: ellipse((0,0),2,1,legend_label="My ellipse", legend_color='green')
```

CHAPTER

FOURTEEN

GRAPH PLOTTING

(For LaTeX drawings of graphs, see the `graph_latex` *module.)*

All graphs have an associated Sage graphics object, which you can display:

```
sage: G = graphs.WheelGraph(15)
sage: P = G.plot()
sage: P.show() # long time
```

If you create a graph in Sage using the `Graph` command, then plot that graph, the positioning of nodes is determined using the spring-layout algorithm. For the special graph constructors, which you get using `graphs.[tab]`, the positions are preset. For example, consider the Petersen graph with default node positioning vs. the Petersen graph constructed by this database:

```
sage: petersen_spring = Graph({0:[1,4,5], 1:[0,2,6], 2:[1,3,7], 3:[2,4,8], 4:[0,3,9], 5:[0,7,8], 6:[1
sage: petersen_spring.show() # long time
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

For all the constructors in this database (except the octahedral, dodecahedral, random and empty graphs), the position dictionary is filled in, instead of using the spring-layout algorithm.

Plot options

Here is the list of options accepted by `plot()` and the constructor of `GraphPlot`.

partition	A partition of the vertex set. If specified, plot will show each cell in a different color. vertex_colors takes precedence.
dist	The distance between multiedges.
vertex_labels	Whether or not to draw vertex labels.
edge_color	The default color for edges.
spring	Use spring layout to finalize the current layout.
pos	The position dictionary of vertices
loop_size	The radius of the smallest loop.
color_by_label	Whether to color the edges according to their labels. This also accepts a function or dictionary mapping labels to colors.
iterations	The number of times to execute the spring layout algorithm.
talk	Whether to display the vertices in talk mode (larger and white).
edge_labels	Whether or not to draw edge labels.
vertex_size	The size to draw the vertices.
dim	The dimension of the layout – 2 or 3.
edge_style	The linestyle of the edges. It should be one of "solid", "dashed", "dotted", dashdot", or "-", "–", ":", "-.", respectively. This currently only works for directed graphs, since we pass off the undirected graph to networkx.
layout	A layout algorithm – one of : "acyclic", "circular" (plots the graph with vertices evenly distributed on a circle), "ranked", "graphviz", "planar", "spring" (traditional spring layout, using the graph's current positions as initial positions), or "tree" (the tree will be plotted in levels, depending on minimum distance for the root).
vertex_shape	The shape to draw the vertices. Currently unavailable for Multi-edged DiGraphs.
vertex_colors	Dictionary of vertex coloring : each key is a color recognizable by matplotlib, and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it does not get drawn).
by_component	Whether to do the spring layout by connected component – a boolean.
heights	A dictionary mapping heights to the list of vertices at this height.
graph_border	Whether or not to draw a frame around the graph.
max_dist	The max distance range to allow multiedges.
prog	Which graphviz layout program to use – one of "circo", "dot", "fdp", "neato", or "twopi".
edge_colors	a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges.
tree_orientation	The direction of tree branches – 'up', 'down', 'left' or 'right'.
save_pos	Whether or not to save the computed position for the graph.
tree_root	A vertex designation for drawing trees. A vertex of the tree to be used as the root for the `layout='tree'` option. If no root is specified, then one is chosen close to the center of the tree. Ignored unless `layout='tree'`

Default options

This module defines two dictionaries containing default options for the `plot()` and `show()` methods. These two dictionaries are `sage.graphs.graph_plot.DEFAULT_PLOT_OPTIONS` and `sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS`, respectively.

Obviously, these values are overruled when arguments are given explicitly.

Here is how to define the default size of a graph drawing to be `[6,6]`. The first two calls to `show()` use this option, while the third does not (a value for `figsize` is explicitly given):

```
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [6,6]
sage: graphs.PetersenGraph().show() # long time
sage: graphs.ChvatalGraph().show()  # long time
sage: graphs.PetersenGraph().show(figsize=[4,4]) # long time
```

We can now reset the default to its initial value, and now display graphs as previously:

```
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [4,4]
sage: graphs.PetersenGraph().show() # long time
sage: graphs.ChvatalGraph().show()  # long time
```

Note:

- While `DEFAULT_PLOT_OPTIONS` affects both `G.show()` and `G.plot()`, settings from `DEFAULT_SHOW_OPTIONS` only affects `G.show()`.
- In order to define a default value permanently, you can add a couple of lines to Sage's startup scripts. Example

```
sage: import sage.graphs.graph_plot
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [4,4]
```

Index of methods and functions

`GraphPlot.set_pos()`	Sets the position plotting parameters for this GraphPlot.
`GraphPlot.set_vertices()`	Sets the vertex plotting parameters for this GraphPlot.
`GraphPlot.set_edges()`	Sets the edge (or arrow) plotting parameters for the GraphPlot object.
`GraphPlot.show()`	Shows the (Di)Graph associated with this GraphPlot object.
`GraphPlot.plot()`	Returns a graphics object representing the (di)graph.
`GraphPlot.layout_tree()`	Compute a nice layout of a tree.
`_circle_embedding()`	Sets some vertices on a circle in the embedding of a graph G.
`_line_embedding()`	Sets some vertices on a line in the embedding of a graph G.

14.1 Methods and classes

`sage.graphs.graph_plot.`**`_circle_embedding`**(*g*, *vertices*, *center=(0, 0)*, *radius=1*, *shift=0*)

Sets some vertices on a circle in the embedding of a graph G.

This method modifies the graph's embedding so that the vertices listed in `vertices` appear in this ordering on a circle of given radius and center. The `shift` parameter is actually a rotation of the circle. A value of `shift=1` will replace in the drawing the i-th element of the list by the $(i-1)$-th. Non-integer values are admissible, and a value of α corresponds to a rotation of the circle by an angle of $\alpha 2\pi/n$ (where n is the number of vertices set on the circle).

EXAMPLE:

```
sage: from sage.graphs.graph_plot import _circle_embedding
sage: g = graphs.CycleGraph(5)
sage: _circle_embedding(g, [0, 2, 4, 1, 3], radius=2, shift=.5)
sage: g.show()
```

`sage.graphs.graph_plot.`**`_line_embedding`**(*g*, *vertices*, *first=(0, 0)*, *last=(0, 1)*)

Sets some vertices on a line in the embedding of a graph G.

This method modifies the graph's embedding so that the vertices of `vertices` appear on a line, where the position of `vertices[0]` is the pair `first` and the position of `vertices[-1]` is `last`. The vertices are evenly spaced.

EXAMPLE:

```
sage: from sage.graphs.graph_plot import _line_embedding
sage: g = graphs.PathGraph(5)
sage: _line_embedding(g, [0, 2, 4, 1, 3], first=(-1, -1), last=(1, 1))
sage: g.show()
```

class sage.graphs.graph_plot.**GraphPlot**(*graph*, *options*)
Bases: sage.structure.sage_object.SageObject

Returns a GraphPlot object, which stores all the parameters needed for plotting (Di)Graphs. A GraphPlot has a plot and show function, as well as some functions to set parameters for vertices and edges. This constructor assumes default options are set. Defaults are shown in the example below.

EXAMPLE:

```
sage: from sage.graphs.graph_plot import GraphPlot
sage: options = {
...     'vertex_size':200,
...     'vertex_labels':True,
...     'layout':None,
...     'edge_style':'solid',
...     'edge_color':'black',
...     'edge_colors':None,
...     'edge_labels':False,
...     'iterations':50,
...     'tree_orientation':'down',
...     'heights':None,
...     'graph_border':False,
...     'talk':False,
...     'color_by_label':False,
...     'partition':None,
...     'dist':.075,
...     'max_dist':1.5,
...     'loop_size':.075}
sage: g = Graph({0:[1,2], 2:[3], 4:[0,1]})
sage: GP = GraphPlot(g, options)
```

layout_tree(*root*, *orientation*)
Compute a nice layout of a tree.

INPUT:

- root – the root vertex.
- orientation – Whether to place the root at the top or at the bottom :
  - orientation="down" – children are placed below their parent
  - orientation="top" – children are placed above their parent

EXAMPLES:

```
sage: T = graphs.RandomLobster(25,0.3,0.3)
sage: T.show(layout='tree',tree_orientation='up') # indirect doctest

sage: from sage.graphs.graph_plot import GraphPlot
sage: G = graphs.HoffmanSingletonGraph()
sage: T = Graph()
sage: T.add_edges(G.min_spanning_tree(starting_vertex=0))
sage: T.show(layout='tree',tree_root=0) # indirect doctest
```

plot(***kwds*)
Returns a graphics object representing the (di)graph.

INPUT:

The options accepted by this method are to be found in the documentation of the sage.graphs.graph_plot module, and the show() method.

Note: See the module's documentation for information on default values of this method.

We can specify some pretty precise plotting of familiar graphs:

```
sage: from math import sin, cos, pi
sage: P = graphs.PetersenGraph()
sage: d = {'#FF0000':[0,5], '#FF9900':[1,6], '#FFFF00':[2,7], '#00FF00':[3,8], '#0000FF':[4,
sage: pos_dict = {}
sage: for i in range(5):
...     x = float(cos(pi/2 + ((2*pi)/5)*i))
...     y = float(sin(pi/2 + ((2*pi)/5)*i))
...     pos_dict[i] = [x,y]
...
sage: for i in range(10)[5:]:
...     x = float(0.5*cos(pi/2 + ((2*pi)/5)*i))
...     y = float(0.5*sin(pi/2 + ((2*pi)/5)*i))
...     pos_dict[i] = [x,y]
...
sage: pl = P.graphplot(pos=pos_dict, vertex_colors=d)
sage: pl.show()
```

Here are some more common graphs with typical options:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()

sage: G = graphs.HeawoodGraph().copy(sparse=True)
sage: for u,v,l in G.edges():
...     G.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: G.graphplot(edge_labels=True).show()
```

The options for plotting also work with directed graphs:

```
sage: D = DiGraph( { 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4], 4: [17, 5], 5: [6, 15
sage: for u,v,l in D.edges():
...     D.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: D.graphplot(edge_labels=True, layout='circular').show()
```

This example shows off the coloring of edges:

```
sage: from sage.plot.colors import rainbow
sage: C = graphs.CubeGraph(5)
sage: R = rainbow(5)
sage: edge_colors = {}
sage: for i in range(5):
...     edge_colors[R[i]] = []
sage: for u,v,l in C.edges():
...     for i in range(5):
...         if u[i] != v[i]:
...             edge_colors[R[i]].append((u,v,l))
sage: C.graphplot(vertex_labels=False, vertex_size=0, edge_colors=edge_colors).show()
```

With the `partition` option, we can separate out same-color groups of vertices:

```
sage: D = graphs.DodecahedralGraph()
sage: Pi = [[6,5,15,14,7],[16,13,8,2,4],[12,17,9,3,1],[0,19,18,10,11]]
sage: D.show(partition=Pi)
```

Loops are also plotted correctly:

```
sage: G = graphs.PetersenGraph()
sage: G.allow_loops(True)
sage: G.add_edge(0,0)
sage: G.show()

sage: D = DiGraph({0:[0,1], 1:[2], 2:[3]}, loops=True)
sage: D.show()
sage: D.show(edge_colors={(0,1,0):[(0,1,None),(1,2,None)],(0,0,0):[(2,3,None)]})
```

More options:

```
sage: pos = {0:[0.0, 1.5], 1:[-0.8, 0.3], 2:[-0.6, -0.8], 3:[0.6, -0.8], 4:[0.8, 0.3]}
sage: g = Graph({0:[1], 1:[2], 2:[3], 3:[4], 4:[0]})
sage: g.graphplot(pos=pos, layout='spring', iterations=0).plot()

sage: G = Graph()
sage: P = G.graphplot().plot()
sage: P.axes()
False
sage: G = DiGraph()
sage: P = G.graphplot().plot()
sage: P.axes()
False
```

We can plot multiple graphs:

```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}).plot()

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}).plot()
sage: t.set_edge_label(0,1,-7)
sage: t.set_edge_label(0,5,3)
sage: t.set_edge_label(0,5,99)
sage: t.set_edge_label(1,2,1000)
sage: t.set_edge_label(3,2,'spam')
sage: t.set_edge_label(2,6,3/2)
sage: t.set_edge_label(0,4,66)
sage: t.graphplot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}, edge_labels=True).plot()

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.graphplot(layout='tree').show()
```

The tree layout is also useful:

```
sage: t = DiGraph('JCC???@A??GO??CO??GO??')
sage: t.graphplot(layout='tree', tree_root=0, tree_orientation="up").show()
```

More examples:

```
sage: D = DiGraph({0:[1,2,3], 2:[1,4], 3:[0]})
sage: D.graphplot().show()

sage: D = DiGraph(multiedges=True, sparse=True)
sage: for i in range(5):
...       D.add_edge((i,i+1,'a'))
```

```
...       D.add_edge((i,i-1,'b'))
sage: D.graphplot(edge_labels=True,edge_colors=D._color_by_label()).plot()

sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: g.graphplot(edge_labels=True, color_by_label=True, edge_style='dashed').plot()
```

The `edge_style` option may be provided in the short format too:

```
sage: g.graphplot(edge_labels=True, color_by_label=True, edge_style='--').plot()
```

TESTS:

Make sure that show options work with plot also:

```
sage: g = Graph({})
sage: g.plot(title='empty graph', axes=True)
```

Check for invalid inputs:

```
sage: p = graphs.PetersenGraph().plot(egabrag='garbage')
Traceback (most recent call last):
...
ValueError: Invalid input 'egabrag=garbage'
```

set_edges(***edge_options*)

Sets the edge (or arrow) plotting parameters for the `GraphPlot` object.

This function is called by the constructor but can also be called to make updates to the vertex options of an existing `GraphPlot` object. Note that the changes are cumulative.

EXAMPLES:

```
sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = g.graphplot(vertex_size=100, edge_labels=True, color_by_label=True, edge_style='d
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_edges(edge_color='black')
sage: GP.plot()

sage: d = DiGraph({}, loops=True, multiedges=True, sparse=True)
sage: d.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = d.graphplot(vertex_size=100, edge_labels=True, color_by_label=True, edge_style='d
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_edges(edge_color='black')
sage: GP.plot()
```

TESTS:

```
sage: G = Graph("Fooba")
sage: G.show(edge_colors={'red':[(3,6),(2,5)]})
```

Verify that default edge labels are pretty close to being between the vertices in some cases where they weren't due to truncating division (trac ticket #10124):

```
sage: test_graphs = graphs.FruchtGraph(), graphs.BullGraph()
sage: tol = 0.001
sage: for G in test_graphs:
...       E=G.edges()
...       for e0, e1, elab in E:
...           G.set_edge_label(e0, e1, '%d %d' % (e0, e1))
...       gp = G.graphplot(save_pos=True,edge_labels=True)
...       vx = gp._plot_components['vertices'][0].xdata
...       vy = gp._plot_components['vertices'][0].ydata
...       for elab in gp._plot_components['edge_labels']:
...           textobj = elab[0]
...           x, y, s = textobj.x, textobj.y, textobj.string
...           v0, v1 = map(int, s.split())
...           vn = vector(((x-(vx[v0]+vx[v1])/2.),y-(vy[v0]+vy[v1])/2.)).norm()
...           assert vn < tol
```

set_pos()

Sets the position plotting parameters for this GraphPlot.

EXAMPLES:

This function is called implicitly by the code below:

```
sage: g = Graph({0:[1,2], 2:[3], 4:[0,1]})
sage: g.graphplot(save_pos=True, layout='circular') # indirect doctest
GraphPlot object for Graph on 5 vertices
```

The following illustrates the format of a position dictionary, but due to numerical noise we do not check the values themselves:

```
sage: g.get_pos()
{0: [...e-17, 1.0],
 1: [-0.951..., 0.309...],
 2: [-0.587..., -0.809...],
 3: [0.587..., -0.809...],
 4: [0.951..., 0.309...]}
```

```
sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]})
```

TESTS:

Make sure that vertex locations are floats. Not being floats isn't a bug in itself but makes it too easy to accidentally introduce a bug elsewhere, such as in `set_edges()` (trac ticket #10124), via silent truncating division of integers:

```
sage: g = graphs.FruchtGraph()
sage: gp = g.graphplot()
sage: set(map(type, flatten(gp._pos.values())))
set([<type 'float'>])
sage: g = graphs.BullGraph()
sage: gp = g.graphplot(save_pos=True)
sage: set(map(type, flatten(gp._pos.values())))
set([<type 'float'>])
```

set_vertices(***vertex_options*)

Sets the vertex plotting parameters for this `GraphPlot`. This function is called by the constructor but can also be called to make updates to the vertex options of an existing `GraphPlot` object. Note that the changes are cumulative.

EXAMPLES:

```
sage: g = Graph({}, loops=True, multiedges=True, sparse=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: GP = g.graphplot(vertex_size=100, edge_labels=True, color_by_label=True, edge_style='d
sage: GP.set_vertices(talk=True)
sage: GP.plot()
sage: GP.set_vertices(vertex_colors='pink', vertex_shape='^')
sage: GP.plot()
```

show(***kwds*)

Shows the (Di)Graph associated with this `GraphPlot` object.

INPUT:

This method accepts all parameters of `sage.plot.graphics.Graphics.show()`.

Note:

- See `the module's documentation` for information on default values of this method.
- Any options not used by plot will be passed on to the `show()` method.

EXAMPLE:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

CHAPTER

FIFTEEN

LINE PLOTS

class sage.plot.line.**Line**(*xdata*, *ydata*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive_xydata

Primitive class that initializes the line graphics type.

EXAMPLES:

```
sage: from sage.plot.line import Line
sage: Line([1,2,7], [1,5,-1], {})
Line defined by 3 points
```

plot3d(*z=0*, ***kwds*)

Plots a 2D line in 3D, with default height zero.

EXAMPLES:

```
sage: E = EllipticCurve('37a').plot(thickness=5).plot3d()
sage: F = EllipticCurve('37a').plot(thickness=5).plot3d(z=2)
sage: E + F  # long time (5s on sage.math, 2012)
```

sage.plot.line.**line**(*points*, ***kwds*)

Returns either a 2-dimensional or 3-dimensional line depending on value of points.

INPUT:

- points - either a single point (as a tuple), a list of points, a single complex number, or a list of complex numbers.

For information regarding additional arguments, see either line2d? or line3d?.

EXAMPLES:

```
sage: line([(0,0), (1,1)])

sage: line([(0,0,1), (1,1,1)])
```

sage.plot.line.**line2d**(*points*, *legend_color=None*, *rgbcolor=(0, 0, 1)*, *thickness=1*, *aspect_ratio='automatic'*, *alpha=1*, *legend_label=None*, ***options*)

Create the line through the given list of points.

INPUT:

- points - either a single point (as a tuple), a list of points, a single complex number, or a list of complex numbers.

Type line2d.options for a dictionary of the default options for lines. You can change this to change the defaults for all future lines. Use line2d.reset() to reset to the default options.

INPUT:

- `alpha` – How transparent the line is
- `thickness` – How thick the line is
- `rgbcolor` – The color as an RGB tuple
- `hue` – The color given as a hue
- `legend_color` – The color of the text in the legend
- `legend_label` – the label for this item in the legend

Any MATPLOTLIB line option may also be passed in. E.g.,

- **`linestyle` - (default: "-") The style of the line, which is one of**
  - `"-"` or `"solid"`
  - `"--"` or `"dashed"`
  - `"-."` or `"dash dot"`
  - `":"` or `"dotted"`
  - `"None"` or `" "` or `""` (nothing)

  The linestyle can also be prefixed with a drawing style (e.g., `"steps--"`)
  - `"default"` (connect the points with straight lines)
  - `"steps"` or `"steps-pre"` (step function; horizontal line is to the left of point)
  - `"steps-mid"` (step function; points are in the middle of horizontal lines)
  - `"steps-post"` (step function; horizontal line is to the right of point)
- **`marker` - The style of the markers, which is one of**
  - `"None"` or `" "` or `""` (nothing) – default
  - `","` (pixel), `"."` (point)
  - `"_"` (horizontal line), `"|"` (vertical line)
  - `"o"` (circle), `"p"` (pentagon), `"s"` (square), `"x"` (x), `"+"` (plus), `"*"` (star)
  - `"D"` (diamond), `"d"` (thin diamond)
  - `"H"` (hexagon), `"h"` (alternative hexagon)
  - `"<"` (triangle left), `">"` (triangle right), `"^"` (triangle up), `"v"` (triangle down)
  - `"1"` (tri down), `"2"` (tri up), `"3"` (tri left), `"4"` (tri right)
  - `0` (tick left), `1` (tick right), `2` (tick up), `3` (tick down)
  - `4` (caret left), `5` (caret right), `6` (caret up), `7` (caret down)
  - `"$...$"` (math TeX string)
- `markersize` – the size of the marker in points
- `markeredgecolor` – the color of the marker edge
- `markerfacecolor` – the color of the marker face
- `markeredgewidth` – the size of the marker edge in points

EXAMPLES:

A line with no points or one point:

```
sage: line([])      #returns an empty plot
sage: import numpy; line(numpy.array([]))
sage: line([(1,1)])
```

A line with numpy arrays:

```
sage: line(numpy.array([[1,2], [3,4]]))
```

A line with a legend:

```
sage: line([(0,0),(1,1)], legend_label='line')
```

Lines with different colors in the legend text:

```
sage: p1 = line([(0,0),(1,1)], legend_label='line')
sage: p2 = line([(1,1),(2,4)], legend_label='squared', legend_color='red')
sage: p1 + p2
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: line([(0,1), (3,4)], figsize=[10, 2])
sage: line([(0,1), (3,4)]).show(figsize=[10, 2]) # These are equivalent
```

We can also use a logarithmic scale if the data will support it:

```
sage: line([(1,2),(2,4),(3,4),(4,8),(4.5,32)],scale='loglog',base=2)
```

Many more examples below!

A blue conchoid of Nicomedes:

```
sage: L = [[1+5*cos(pi/2+pi*i/100), tan(pi/2+pi*i/100)*(1+5*cos(pi/2+pi*i/100))] for i in range(
sage: line(L, rgbcolor=(1/4,1/8,3/4))
```

A line with 2 complex points:

```
sage: i = CC.0
sage: line([1+i, 2+3*i])
```

A blue hypotrochoid (3 leaves):

```
sage: n = 4; h = 3; b = 2
sage: L = [[n*cos(pi*i/100)+h*cos((n/b)*pi*i/100),n*sin(pi*i/100)-h*sin((n/b)*pi*i/100)] for i i
sage: line(L, rgbcolor=(1/4,1/4,3/4))
```

A blue hypotrochoid (4 leaves):

```
sage: n = 6; h = 5; b = 2
sage: L = [[n*cos(pi*i/100)+h*cos((n/b)*pi*i/100),n*sin(pi*i/100)-h*sin((n/b)*pi*i/100)] for i i
sage: line(L, rgbcolor=(1/4,1/4,3/4))
```

A red limacon of Pascal:

```
sage: L = [[sin(pi*i/100)+sin(pi*i/50),-(1+cos(pi*i/100)+cos(pi*i/50))] for i in range(-100,101)
sage: line(L, rgbcolor=(1,1/4,1/2))
```

A light green trisectrix of Maclaurin:

```
sage: L = [[2*(1-4*cos(-pi/2+pi*i/100)^2),10*tan(-pi/2+pi*i/100)*(1-4*cos(-pi/2+pi*i/100)^2)] fo
sage: line(L, rgbcolor=(1/4,1,1/8))
```

A green lemniscate of Bernoulli:

```
sage: cosines = [cos(-pi/2+pi*i/100) for i in range(201)]
sage: v = [(1/c, tan(-pi/2+pi*i/100)) for i,c in enumerate(cosines) if c != 0]
sage: L = [(a/(a^2+b^2), b/(a^2+b^2)) for a,b in v]
sage: line(L, rgbcolor=(1/4,3/4,1/8))
```

A red plot of the Jacobi elliptic function $\mathrm{sn}(x, 2)$, $-3 < x < 3$:

```
sage: L = [(i/100.0, real_part(jacobi('sn', i/100.0, 2.0))) for i in
....:      range(-300, 300, 30)]
sage: line(L, rgbcolor=(3/4, 1/4, 1/8))
```

A red plot of J-Bessel function $J_2(x)$, $0 < x < 10$:

```
sage: L = [(i/10.0, bessel_J(2,i/10.0)) for i in range(100)]
sage: line(L, rgbcolor=(3/4,1/4,5/8))
```

A purple plot of the Riemann zeta function $\zeta(1/2 + it)$, $0 < t < 30$:

```
sage: i = CDF.gen()
sage: v = [zeta(0.5 + n/10 * i) for n in range(300)]
sage: L = [(z.real(), z.imag()) for z in v]
sage: line(L, rgbcolor=(3/4,1/2,5/8))
```

A purple plot of the Hasse-Weil L-function $L(E, 1 + it)$, $-1 < t < 10$:

```
sage: E = EllipticCurve('37a')
sage: vals = E.lseries().values_along_line(1-I, 1+10*I, 100) # critical line
sage: L = [(z[1].real(), z[1].imag()) for z in vals]
sage: line(L, rgbcolor=(3/4,1/2,5/8))
```

A red, blue, and green "cool cat":

```
sage: G = plot(-cos(x), -2, 2, thickness=5, rgbcolor=(0.5,1,0.5))
sage: P = polygon([[1,2], [5,6], [5,0]], rgbcolor=(1,0,0))
sage: Q = polygon([(-x,y) for x,y in P[0]], rgbcolor=(0,0,1))
sage: G + P + Q   # show the plot
```

TESTS:

Check that trac ticket #13690 is fixed. The legend label should have circles as markers.:

```
sage: line(enumerate(range(2)), marker='o', legend_label='circle')
```

CHAPTER

SIXTEEN

STEP FUNCTION PLOTS

sage.plot.step.**plot_step_function**(*v*, *vertical_lines=True*, ***kwds*)

Return the line graphics object that gives the plot of the step function f defined by the list v of pairs (a, b). Here if (a, b) is in v, then $f(a) = b$. The user does not have to worry about sorting the input list v.

INPUT:

- v – list of pairs (a,b)
- vertical_lines – bool (default: True) if True, draw vertical risers at each step of this step function. Technically these vertical lines are not part of the graph of this function, but they look very nice in the plot so we include them by default

EXAMPLES:

We plot the prime counting function:

```
sage: plot_step_function([(i,prime_pi(i)) for i in range(20)])

sage: plot_step_function([(i,sin(i)) for i in range(5,20)])
```

We pass in many options and get something that looks like "Space Invaders":

```
sage: v = [(i,sin(i)) for i in range(5,20)]
sage: plot_step_function(v, vertical_lines=False, thickness=30, rgbcolor='purple', axes=False)
```

CHAPTER

SEVENTEEN

MATRIX PLOTS

class sage.plot.matrix_plot.**MatrixPlot**(*xy_data_array*, *xrange*, *yrange*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class for the matrix plot graphics type. See matrix_plot? for help actually doing matrix plots.

INPUT:

- xy_data_array - list of lists giving matrix values corresponding to the grid
- xrange - tuple of 2 floats indicating range for horizontal direction (number of columns in the matrix)
- yrange - tuple of 2 floats indicating range for vertical direction (number of rows in the matrix)
- options - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via matrix_plot():

```
sage: from sage.plot.matrix_plot import MatrixPlot
sage: M = MatrixPlot([[1,3],[2,4]],(1,2),(2,3),options={'cmap':'winter'})
sage: M
MatrixPlot defined by a 2 x 2 data grid
sage: M.yrange
(2, 3)
sage: M.xy_data_array
[[1, 3], [2, 4]]
sage: M.options()
{'cmap': 'winter'}
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: matrix_plot([[1, 0], [0, 1]], fontsize=10)
sage: matrix_plot([[1, 0], [0, 1]]).show(fontsize=10) # These are equivalent
```

TESTS:

We test creating a matrix plot:

```
sage: matrix_plot([[mod(i,5)^j for i in range(5)] for j in range(1,6)])
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: m = matrix_plot(matrix([[1,3,5,1],[2,4,5,6],[1,3,5,7]]))[0]
sage: list(sorted(m.get_minmax_data().items()))
[('xmax', 3.5), ('xmin', -0.5), ('ymax', -0.5), ('ymin', 2.5)]
```

sage.plot.matrix_plot.**matrix_plot**(*mat*, *origin='upper'*, *vmin=None*, *frame=True*, *axes=False*, *colorbar=False*, *vmax=None*, *subdivisions=False*, *cmap='gray'*, *ticks_integer=True*, *marker='.'*, *norm=None*, *subdivision_boundaries=None*, *subdivision_style=None*, *colorbar_orientation='vertical'*, *colorbar_format=None*, ***options*)

A plot of a given matrix or 2D array.

If the matrix is dense, each matrix element is given a different color value depending on its relative size compared to the other elements in the matrix. If the matrix is sparse, colors only indicate whether an element is nonzero or zero, so the plot represents the sparsity pattern of the matrix.

The tick marks drawn on the frame axes denote the row numbers (vertical ticks) and the column numbers (horizontal ticks) of the matrix.

INPUT:

- `mat` - a 2D matrix or array

The following input must all be passed in as named parameters, if default not used:

- `cmap` - a colormap (default: 'gray'), the name of a predefined colormap, a list of colors, or an instance of a matplotlib Colormap. Type: `import matplotlib.cm; matplotlib.cm.datad.keys()` for available colormap names.
- `colorbar` – boolean (default: False) Show a colorbar or not (dense matrices only).

  The following options are used to adjust the style and placement of colorbars. They have no effect if a colorbar is not shown.

  - `colorbar_orientation` – string (default: 'vertical'), controls placement of the colorbar, can be either 'vertical' or 'horizontal'
  - `colorbar_format` – a format string, this is used to format the colorbar labels.
  - `colorbar_options` – a dictionary of options for the matplotlib colorbar API. Documentation for the `matplotlib.colorbar` module has details.
- `norm` - If None (default), the value range is scaled to the interval [0,1]. If 'value', then the actual value is used with no scaling. A `matplotlib.colors.Normalize` instance may also passed.
- `vmin` - The minimum value (values below this are set to this value)
- `vmax` - The maximum value (values above this are set to this value)
- `origin` - If 'upper' (default), the first row of the matrix is on the top of the graph. If 'lower', the first row is on the bottom of the graph.
- `subdivisions` - If True, plot the subdivisions of the matrix as lines.
- `subdivision_boundaries` - a list of lists in the form `[row_subdivisions, column_subdivisions]`, which specifies the row and column subdivisions to use. If not specified, defaults to the matrix subdivisions
- `subdivision_style` - a dictionary of properties passed on to the `line2d()` command for plotting subdivisions. If this is a two-element list or tuple, then it specifies the styles of row and column divisions, respectively.

EXAMPLES:

A matrix over **Z** colored with different grey levels:

```
sage: matrix_plot(matrix([[1,3,5,1],[2,4,5,6],[1,3,5,7]]))
```

Here we make a random matrix over $\mathbf{R}$ and use `cmap='hsv'` to color the matrix elements different RGB colors:

```
sage: matrix_plot(random_matrix(RDF, 50), cmap='hsv')
```

By default, entries are scaled to the interval [0,1] before determining colors from the color map. That means the two plots below are the same:

```
sage: P = matrix_plot(matrix(2,[1,1,3,3]))
sage: Q = matrix_plot(matrix(2,[2,2,3,3]))
sage: P; Q
```

However, we can specify which values scale to 0 or 1 with the `vmin` and `vmax` parameters (values outside the range are clipped). The two plots below are now distinguished:

```
sage: P = matrix_plot(matrix(2,[1,1,3,3]), vmin=0, vmax=3, colorbar=True)
sage: Q = matrix_plot(matrix(2,[2,2,3,3]), vmin=0, vmax=3, colorbar=True)
sage: P; Q
```

We can also specify a norm function of 'value', which means that there is no scaling performed:

```
sage: matrix_plot(random_matrix(ZZ,10)*.05, norm='value', colorbar=True)
```

Matrix subdivisions can be plotted as well:

```
sage: m=random_matrix(RR,10)
sage: m.subdivide([2,4],[6,8])
sage: matrix_plot(m, subdivisions=True, subdivision_style=dict(color='red',thickness=3))
```

You can also specify your own subdivisions and separate styles for row or column subdivisions:

```
sage: m=random_matrix(RR,10)
sage: matrix_plot(m, subdivisions=True, subdivision_boundaries=[[2,4],[6,8]], subdivision_style=
```

Generally matrices are plotted with the (0,0) entry in the upper left. However, sometimes if we are plotting an image, we'd like the (0,0) entry to be in the lower left. We can do that with the `origin` argument:

```
sage: matrix_plot(identity_matrix(100), origin='lower')
```

Another random plot, but over $\mathbf{F}_{389}$:

```
sage: m = random_matrix(GF(389), 10)
sage: matrix_plot(m, cmap='Oranges')
```

It also works if you lift it to the polynomial ring:

```
sage: matrix_plot(m.change_ring(GF(389)['x']), cmap='Oranges')
```

We have several options for colorbars:

```
sage: matrix_plot(random_matrix(RDF, 50), colorbar=True, colorbar_orientation='horizontal')
```

```
sage: matrix_plot(random_matrix(RDF, 50), colorbar=True, colorbar_format='%.3f')
```

The length of a color bar and the length of the adjacent matrix plot dimension may be quite different. This example shows how to adjust the length of the colorbar by passing a dictionary of options to the matplotlib colorbar routines.

```
sage: m = random_matrix(ZZ, 40, 80, x=-10, y=10)
sage: m.plot(colorbar=True, colorbar_orientation='vertical',
...           colorbar_options={'shrink':0.50})
```

Here we plot a random sparse matrix:

```
sage: sparse = matrix(dict([((randint(0, 10), randint(0, 10)), 1) for i in xrange(100)]))
sage: matrix_plot(sparse)

sage: A=random_matrix(ZZ,100000,density=.00001,sparse=True)
sage: matrix_plot(A,marker=',')
```

As with dense matrices, sparse matrix entries are automatically converted to floating point numbers before plotting. Thus the following works:

```
sage: b=random_matrix(GF(2),200,sparse=True,density=0.01)
sage: matrix_plot(b)
```

While this returns an error:

```
sage: b=random_matrix(CDF,200,sparse=True,density=0.01)
sage: matrix_plot(b)
Traceback (most recent call last):
...
ValueError: can not convert entries to floating point numbers
```

To plot the absolute value of a complex matrix, use the `apply_map` method:

```
sage: b=random_matrix(CDF,200,sparse=True,density=0.01)
sage: matrix_plot(b.apply_map(abs))
```

Plotting lists of lists also works:

```
sage: matrix_plot([[1,3,5,1],[2,4,5,6],[1,3,5,7]])
```

As does plotting of NumPy arrays:

```
sage: import numpy
sage: matrix_plot(numpy.random.rand(10, 10))
```

A plot title can be added to the matrix plot.:

```
sage: matrix_plot(identity_matrix(50), origin='lower', title='not identity')
```

The title position is adjusted upwards if the `origin` keyword is set to `"upper"` (this is the default).:

```
sage: matrix_plot(identity_matrix(50), title='identity')
```

TESTS:

```
sage: P.<t> = RR[]
sage: matrix_plot(random_matrix(P, 3, 3))
Traceback (most recent call last):
...
TypeError: cannot coerce nonconstant polynomial to float

sage: matrix_plot([1,2,3])
Traceback (most recent call last):
...
TypeError: mat must be a Matrix or a two dimensional array

sage: matrix_plot([[sin(x), cos(x)], [1, 0]])
Traceback (most recent call last):
...
TypeError: mat must be a Matrix or a two dimensional array
```

Test that sparse matrices also work with subdivisions:

```
sage: matrix_plot(sparse, subdivisions=True, subdivision_boundaries=[[2,4],[6,8]])
```

CHAPTER

EIGHTEEN

PLOTTING FIELDS

class sage.plot.plot_field.**PlotField**(*xpos_array*, *ypos_array*, *xvec_array*, *yvec_array*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Primitive class that initializes the PlotField graphics type

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: d = plot_vector_field((.01*x,x+y), (x,10,20), (y,10,20))[0].get_minmax_data()
sage: d['xmin']
10.0
sage: d['ymin']
10.0
```

sage.plot.plot_field.**plot_slope_field**(*f*, *xrange*, *yrange*, ***kwds*)

`plot_slope_field` takes a function of two variables xvar and yvar (for instance, if the variables are x and y, take $f(x, y)$), and at representative points (x_i, y_i) between xmin, xmax, and ymin, ymax respectively, plots a line with slope $f(x_i, y_i)$ (see below).

```
plot_slope_field(f, (xvar, xmin, xmax), (yvar, ymin, ymax))
```

EXAMPLES:

A logistic function modeling population growth:

```
sage: x,y = var('x y')
sage: capacity = 3 # thousand
sage: growth_rate = 0.7 # population increases by 70% per unit of time
sage: plot_slope_field(growth_rate*(1-y/capacity)*y, (x,0,5), (y,0,capacity*2))
```

Plot a slope field involving sin and cos:

```
sage: x,y = var('x y')
sage: plot_slope_field(sin(x+y)+cos(x+y), (x,-3,3), (y,-3,3))
```

Plot a slope field using a lambda function:

```
sage: plot_slope_field(lambda x,y: x+y, (-2,2), (-2,2))
```

TESTS:

Verify that we're not getting warnings due to use of headless quivers (trac #11208):

```
sage: x,y = var('x y')
sage: import numpy # bump warnings up to errors for testing purposes
sage: old_err = numpy.seterr('raise')
sage: plot_slope_field(sin(x+y)+cos(x+y), (x,-3,3), (y,-3,3))
sage: dummy_err = numpy.seterr(**old_err)
```

sage.plot.plot_field.**plot_vector_field**(*f_g*, *xrange*, *yrange*, *frame=True*, *plot_points=20*, ***options*)

plot_vector_field takes two functions of two variables xvar and yvar (for instance, if the variables are x and y, take $(f(x,y), g(x,y))$) and plots vector arrows of the function over the specified ranges, with xrange being of xvar between xmin and xmax, and yrange similarly (see below).

```
plot_vector_field((f, g), (xvar, xmin, xmax), (yvar, ymin, ymax))
```

EXAMPLES:

Plot some vector fields involving sin and cos:

```
sage: x,y = var('x y')
sage: plot_vector_field((sin(x), cos(y)), (x,-3,3), (y,-3,3))

sage: plot_vector_field(( y, (cos(x)-2)*sin(x)), (x,-pi,pi), (y,-pi,pi))
```

Plot a gradient field:

```
sage: u,v = var('u v')
sage: f = exp(-(u^2+v^2))
sage: plot_vector_field(f.gradient(), (u,-2,2), (v,-2,2), color='blue')
```

Plot two orthogonal vector fields:

```
sage: x,y = var('x,y')
sage: a=plot_vector_field((x,y), (x,-3,3),(y,-3,3),color='blue')
sage: b=plot_vector_field((y,-x),(x,-3,3),(y,-3,3),color='red')
sage: show(a+b)
```

We ignore function values that are infinite or NaN:

```
sage: x,y = var('x,y')
sage: plot_vector_field( (-x/sqrt(x^2+y^2), -y/sqrt(x^2+y^2)), (x, -10, 10), (y, -10, 10))

sage: x,y = var('x,y')
sage: plot_vector_field( (-x/sqrt(x+y), -y/sqrt(x+y)), (x, -10, 10), (y, -10, 10))
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: plot_vector_field((x, y), (x, -2, 2), (y, -2, 2), xmax=10)
sage: plot_vector_field((x, y), (x, -2, 2), (y, -2, 2)).show(xmax=10) # These are equivalent
```

CHAPTER

NINETEEN

POINTS

TESTS:

```
sage: E = EllipticCurve('37a')
sage: P = E(0,0)
sage: def get_points(n): return sum([point(list(i*P)[:2], size=3) for i in range(-n,n) if i != 0 and
sage: sum([get_points(15*n).plot3d(z=n) for n in range(1,10)])
```

class sage.plot.point.**Point**(*xdata*, *ydata*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive_xydata

Primitive class for the point graphics type. See point?, point2d? or point3d? for information about actually plotting points.

INPUT:

- xdata - list of x values for points in Point object
- ydata - list of y values for points in Point object
- options - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via `point` and friends:

```
sage: from sage.plot.point import Point
sage: P = Point([1,2],[2,3],{'alpha':.5})
sage: P
Point set defined by 2 point(s)
sage: P.options()['alpha']
0.500000000000000
sage: P.xdata
[1, 2]
```

TESTS:

We test creating a point:

```
sage: point((3,3))
```

plot3d(*z=0*, ***kwds*)

Plots a two-dimensional point in 3-D, with default height zero.

INPUT:

- z - optional 3D height above xy-plane. May be a list if self is a list of points.

EXAMPLES:

One point:

```
sage: A=point((1,1))
sage: a=A[0];a
Point set defined by 1 point(s)
sage: b=a.plot3d()
```

One point with a height:

```
sage: A=point((1,1))
sage: a=A[0];a
Point set defined by 1 point(s)
sage: b=a.plot3d(z=3)
sage: b.loc[2]
3.0
```

Multiple points:

```
sage: P=point([(0,0), (1,1)])
sage: p=P[0]; p
Point set defined by 2 point(s)
sage: q=p.plot3d(size=22)
```

Multiple points with different heights:

```
sage: P=point([(0,0), (1,1)])
sage: p=P[0]
sage: q=p.plot3d(z=[2,3])
sage: q.all[0].loc[2]
2.0
sage: q.all[1].loc[2]
3.0
```

Note that keywords passed must be valid point3d options:

```
sage: A=point((1,1),size=22)
sage: a=A[0];a
Point set defined by 1 point(s)
sage: b=a.plot3d()
sage: b.size
22
sage: b=a.plot3d(pointsize=23) # only 2D valid option
sage: b.size
22
sage: b=a.plot3d(size=23) # correct keyword
sage: b.size
23
```

TESTS:

Heights passed as a list should have same length as number of points:

```
sage: P=point([(0,0), (1,1), (2,3)])
sage: p=P[0]
sage: q=p.plot3d(z=2)
sage: q.all[1].loc[2]
2.0
sage: q=p.plot3d(z=[2,-2])
Traceback (most recent call last):
...
```

```
ValueError: Incorrect number of heights given
```

sage.plot.point.**point**(*points*, ***kwds*)

Returns either a 2-dimensional or 3-dimensional point or sum of points.

INPUT:

- `points` - either a single point (as a tuple), a list of points, a single complex number, or a list of complex numbers.

For information regarding additional arguments, see either point2d? or point3d?.

EXAMPLES:

```
sage: point((1,2))

sage: point((1,2,3))

sage: point([(0,0), (1,1)])

sage: point([(0,0,1), (1,1,1)])
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)], frame=True)
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)]).show(frame=True) # Th
```

sage.plot.point.**point2d**(*points*, *legend_color=None*, *faceted=False*, *rgbcolor=(0, 0, 1)*, *aspect_ratio='automatic'*, *alpha=1*, *legend_label=None*, *size=10*, ***options*)

A point of size `size` defined by point = (x, y).

INPUT:

- `points` - either a single point (as a tuple), a list of points, a single complex number, or a list of complex numbers.

Type `point2d.options` to see all options.

EXAMPLES:

A purple point from a single tuple or coordinates:

```
sage: point((0.5, 0.5), rgbcolor=hue(0.75))
```

Passing an empty list returns an empty plot:

```
sage: point([])
sage: import numpy; point(numpy.array([]))
```

If you need a 2D point to live in 3-space later, this is possible:

```
sage: A=point((1,1))
sage: a=A[0];a
Point set defined by 1 point(s)
sage: b=a.plot3d(z=3)
```

This is also true with multiple points:

```
sage: P=point([(0,0), (1,1)])
sage: p=P[0]
sage: q=p.plot3d(z=[2,3])
```

Here are some random larger red points, given as a list of tuples:

```
sage: point(((0.5, 0.5), (1, 2), (0.5, 0.9), (-1, -1)), rgbcolor=hue(1), size=30)
```

And an example with a legend:

```
sage: point((0,0), rgbcolor='black', pointsize=40, legend_label='origin')
```

The legend can be colored:

```
sage: P = points([(0,0),(1,0)], pointsize=40, legend_label='origin', legend_color='red')
sage: P + plot(x^2,(x,0,1), legend_label='plot', legend_color='green')
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)], frame=True)
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)]).show(frame=True) # Th
```

For plotting data, we can use a logarithmic scale, as long as we are sure not to include any nonpositive points in the logarithmic direction:

```
sage: point([(1,2),(2,4),(3,4),(4,8),(4.5,32)],scale='semilogy',base=2)
```

Since Sage Version 4.4 (ticket #8599), the size of a 2d point can be given by the argument `size` instead of `pointsize`. The argument `pointsize` is still supported:

```
sage: point((3,4), size=100)
```

```
sage: point((3,4), pointsize=100)
```

We can plot a single complex number:

```
sage: point(CC(1+I), pointsize=100)
```

We can also plot a list of complex numbers:

```
sage: point([CC(I), CC(I+1), CC(2+2*I)], pointsize=100)
```

sage.plot.point.**points**(*points*, ***kwds*)

Returns either a 2-dimensional or 3-dimensional point or sum of points.

INPUT:

- `points` - either a single point (as a tuple), a list of points, a single complex number, or a list of complex numbers.

For information regarding additional arguments, see either point2d? or point3d?.

EXAMPLES:

```
sage: point((1,2))
```

```
sage: point((1,2,3))
```

```
sage: point([(0,0), (1,1)])
```

```
sage: point([(0,0,1), (1,1,1)])
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)], frame=True)
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)]).show(frame=True) # Th
```

CHAPTER

TWENTY

POLYGONS

class sage.plot.polygon.**Polygon**(*xdata*, *ydata*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive_xydata

Primitive class for the Polygon graphics type. For information on actual plotting, please see polygon(), polygon2d(), or polygon3d().

INPUT:

- xdata - list of x-coordinates of points defining Polygon
- ydata - list of y-coordinates of points defining Polygon
- options - dict of valid plot options to pass to constructor

EXAMPLES:

Note this should normally be used indirectly via polygon():

```
sage: from sage.plot.polygon import Polygon
sage: P = Polygon([1,2,3],[2,3,2],{'alpha':.5})
sage: P
Polygon defined by 3 points
sage: P.options()['alpha']
0.500000000000000
sage: P.ydata
[2, 3, 2]
```

TESTS:

We test creating polygons:

```
sage: polygon([(0,0), (1,1), (0,1)])

sage: polygon([(0,0,1), (1,1,1), (2,0,1)])
```

plot3d(*z=0*, ***kwds*)

Plots a 2D polygon in 3D, with default height zero.

INPUT:

- z - optional 3D height above xy-plane, or a list of heights corresponding to the list of 2D polygon points.

EXAMPLES:

A pentagon:

```
sage: polygon([(cos(t), sin(t)) for t in srange(0, 2*pi, 2*pi/5)]).plot3d()
```

Showing behavior of the optional parameter z:

```
sage: P = polygon([(0,0), (1,2), (0,1), (-1,2)])
sage: p = P[0]; p
Polygon defined by 4 points
sage: q = p.plot3d()
sage: q.obj_repr(q.testing_render_params())[2]
['v 0 0 0', 'v 1 2 0', 'v 0 1 0', 'v -1 2 0']
sage: r = p.plot3d(z=3)
sage: r.obj_repr(r.testing_render_params())[2]
['v 0 0 3', 'v 1 2 3', 'v 0 1 3', 'v -1 2 3']
sage: s = p.plot3d(z=[0,1,2,3])
sage: s.obj_repr(s.testing_render_params())[2]
['v 0 0 0', 'v 1 2 1', 'v 0 1 2', 'v -1 2 3']
```

TESTS:

Heights passed as a list should have same length as number of points:

```
sage: P = polygon([(0,0), (1,2), (0,1), (-1,2)])
sage: p = P[0]
sage: q = p.plot3d(z=[2,-2])
Traceback (most recent call last):
...
ValueError: Incorrect number of heights given
```

sage.plot.polygon.**polygon**(*points*, ***options*)

Returns either a 2-dimensional or 3-dimensional polygon depending on value of points.

For information regarding additional arguments, see either `polygon2d()` or `polygon3d()`. Options may be found and set using the dictionaries `polygon2d.options` and `polygon3d.options`.

EXAMPLES:

```
sage: polygon([(0,0), (1,1), (0,1)])
sage: polygon([(0,0,1), (1,1,1), (2,0,1)])
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: polygon([(0,0), (1,1), (0,1)], axes=False)
sage: polygon([(0,0), (1,1), (0,1)]).show(axes=False) # These are equivalent
```

sage.plot.polygon.**polygon2d**(*points*, *legend_color=None*, *rgbcolor=(0, 0, 1)*, *thickness=None*, *aspect_ratio=1.0*, *alpha=1*, *legend_label=None*, *fill=True*, ***options*)

Returns a 2-dimensional polygon defined by `points`.

Type `polygon2d.options` for a dictionary of the default options for polygons. You can change this to change the defaults for all future polygons. Use `polygon2d.reset()` to reset to the default options.

EXAMPLES:

We create a purple-ish polygon:

```
sage: polygon2d([[1,2], [5,6], [5,0]], rgbcolor=(1,0,1))
```

By default, polygons are filled in, but we can make them without a fill as well:

```
sage: polygon2d([[1,2], [5,6], [5,0]], fill=False)
```

In either case, the thickness of the border can be controlled:

```
sage: polygon2d([[1,2], [5,6], [5,0]], fill=False, thickness=4, color='orange')
```

Some modern art – a random polygon, with legend:

```
sage: v = [(randrange(-5,5), randrange(-5,5)) for _ in range(10)]
sage: polygon2d(v, legend_label='some form')
```

A purple hexagon:

```
sage: L = [[cos(pi*i/3),sin(pi*i/3)] for i in range(6)]
sage: polygon2d(L, rgbcolor=(1,0,1))
```

A green deltoid:

```
sage: L = [[-1+cos(pi*i/100)*(1+cos(pi*i/100)),2*sin(pi*i/100)*(1-cos(pi*i/100))] for i in range
sage: polygon2d(L, rgbcolor=(1/8,3/4,1/2))
```

A blue hypotrochoid:

```
sage: L = [[6*cos(pi*i/100)+5*cos((6/2)*pi*i/100),6*sin(pi*i/100)-5*sin((6/2)*pi*i/100)] for i i
sage: polygon2d(L, rgbcolor=(1/8,1/4,1/2))
```

Another one:

```
sage: n = 4; h = 5; b = 2
sage: L = [[n*cos(pi*i/100)+h*cos((n/b)*pi*i/100),n*sin(pi*i/100)-h*sin((n/b)*pi*i/100)] for i i
sage: polygon2d(L, rgbcolor=(1/8,1/4,3/4))
```

A purple epicycloid:

```
sage: m = 9; b = 1
sage: L = [[m*cos(pi*i/100)+b*cos((m/b)*pi*i/100),m*sin(pi*i/100)-b*sin((m/b)*pi*i/100)] for i i
sage: polygon2d(L, rgbcolor=(7/8,1/4,3/4))
```

A brown astroid:

```
sage: L = [[cos(pi*i/100)^3,sin(pi*i/100)^3] for i in range(200)]
sage: polygon2d(L, rgbcolor=(3/4,1/4,1/4))
```

And, my favorite, a greenish blob:

```
sage: L = [[cos(pi*i/100)*(1+cos(pi*i/50)), sin(pi*i/100)*(1+sin(pi*i/50))] for i in range(200)]
sage: polygon2d(L, rgbcolor=(1/8, 3/4, 1/2))
```

This one is for my wife:

```
sage: L = [[sin(pi*i/100)+sin(pi*i/50),-(1+cos(pi*i/100)+cos(pi*i/50))] for i in range(-100,100)
sage: polygon2d(L, rgbcolor=(1,1/4,1/2))
```

One can do the same one with a colored legend label:

```
sage: polygon2d(L, color='red', legend_label='For you!', legend_color='red')
```

Polygons have a default aspect ratio of 1.0:

```
sage: polygon2d([[1,2], [5,6], [5,0]]).aspect_ratio()
1.0
```

AUTHORS:

- David Joyner (2006-04-14): the long list of examples above.

CHAPTER

TWENTYONE

PLOTTING PRIMITIVES

class sage.plot.primitive.**GraphicPrimitive**(*options*)

Bases: sage.structure.sage_object.SageObject

Base class for graphics primitives, e.g., things that knows how to draw themselves in 2D.

EXAMPLES:

We create an object that derives from GraphicPrimitive:

```
sage: P = line([(-1,-2), (3,5)])
sage: P[0]
Line defined by 2 points
sage: type(P[0])
<class 'sage.plot.line.Line'>
```

options()

Return the dictionary of options for this graphics primitive.

By default this function verifies that the options are all valid; if any aren't, then a verbose message is printed with level 0.

EXAMPLES:

```
sage: from sage.plot.primitive import GraphicPrimitive
sage: GraphicPrimitive({}).options()
{}
```

plot3d(***kwds*)

Plots 3D version of 2D graphics object. Not implemented for base class.

EXAMPLES:

```
sage: from sage.plot.primitive import GraphicPrimitive
sage: G=GraphicPrimitive({})
sage: G.plot3d()
Traceback (most recent call last):
...
NotImplementedError: 3D plotting not implemented for Graphics primitive
```

set_options(*new_options*)

Change the options to $new_options$.

EXAMPLES:

```
sage: from sage.plot.circle import Circle
sage: c = Circle(0,0,1,{})
sage: c.set_options({'thickness': 0.6})
```

```
sage: c.options()
{'thickness': 0.6...}
```

set_zorder(*zorder*)

Set the layer in which to draw the object.

EXAMPLES:

```
sage: P = line([(-2,-3), (3,4)], thickness=4)
sage: p=P[0]
sage: p.set_zorder(2)
sage: p.options()['zorder']
2
sage: Q = line([(-2,-4), (3,5)], thickness=4,zorder=1,hue=.5)
sage: P+Q # blue line on top
sage: q=Q[0]
sage: q.set_zorder(3)
sage: P+Q # teal line on top
sage: q.options()['zorder']
3
```

class sage.plot.primitive.**GraphicPrimitive_xydata**(*options*)

Bases: sage.plot.primitive.GraphicPrimitive

Create a base class GraphicsPrimitive. All this does is set the options.

EXAMPLES:

We indirectly test this function:

```
sage: from sage.plot.primitive import GraphicPrimitive
sage: GraphicPrimitive({})
Graphics primitive
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: d = polygon([[1,2], [5,6], [5,0]], rgbcolor=(1,0,1))[0].get_minmax_data()
sage: d['ymin']
0.0
sage: d['xmin']
1.0

sage: d = point((3, 3), rgbcolor=hue(0.75))[0].get_minmax_data()
sage: d['xmin']
3.0
sage: d['ymin']
3.0

sage: l = line([(100, 100), (120, 120)])[0]
sage: d = l.get_minmax_data()
sage: d['xmin']
100.0
sage: d['xmax']
120.0
```

CHAPTER

TWENTYTWO

SCATTER PLOTS

class sage.plot.scatter_plot.**ScatterPlot**(*xdata*, *ydata*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Scatter plot graphics primitive.

Input consists of two lists/arrays of the same length, whose values give the horizontal and vertical coordinates of each point in the scatter plot. Options may be passed in dictionary format.

EXAMPLES:

```
sage: from sage.plot.scatter_plot import ScatterPlot
sage: ScatterPlot([0,1,2], [3.5,2,5.1], {'facecolor':'white', 'marker':'s'})
Scatter plot graphics primitive on 3 data points
```

get_minmax_data()

Returns a dictionary with the bounding box data.

EXAMPLES:

```
sage: s = scatter_plot([[0,1],[2,4],[3.2,6]])
sage: d = s.get_minmax_data()
sage: d['xmin']
0.0
sage: d['ymin']
1.0
```

sage.plot.scatter_plot.**scatter_plot**(*datalist*, *edgecolor='black'*, *facecolor='#fec7b8'*, *clip=True*, *markersize=50*, *aspect_ratio='automatic'*, *zorder=5*, *marker='o'*, *alpha=1*, ***options*)

Returns a Graphics object of a scatter plot containing all points in the datalist. Type scatter_plot.options to see all available plotting options.

INPUT:

- datalist – a list of tuples (x,y)
- alpha – default: 1
- markersize – default: 50
- **marker - The style of the markers (default "o"), which is one of**
  - "None" or " " or "" (nothing)
  - "," (pixel), "." (point)
  - "_" (horizontal line), "|" (vertical line)
  - "o" (circle), "p" (pentagon), "s" (square), "x" (x), "+" (plus), "*" (star)

  - "D" (diamond), "d" (thin diamond)
  - "H" (hexagon), "h" (alternative hexagon)
  - "<" (triangle left), ">" (triangle right), "^" (triangle up), "v" (triangle down)
  - "1" (tri down), "2" (tri up), "3" (tri left), "4" (tri right)
  - 0 (tick left), 1 (tick right), 2 (tick up), 3 (tick down)
  - 4 (caret left), 5 (caret right), 6 (caret up), 7 (caret down)
  - "$...$" (math TeX string)

- facecolor – default: '#fec7b8'
- edgecolor – default: 'black'
- zorder – default: 5

EXAMPLES:

```
sage: scatter_plot([[0,1],[2,2],[4.3,1.1]], marker='s')
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: scatter_plot([(0, 0), (1, 1)], markersize=100, facecolor='green', ymax=100)
sage: scatter_plot([(0, 0), (1, 1)], markersize=100, facecolor='green').show(ymax=100) # These a
```

CHAPTER

TWENTYTHREE

TEXT IN PLOTS

class sage.plot.text.**Text**(*string*, *point*, *options*)

Bases: sage.plot.primitive.GraphicPrimitive

Base class for Text graphics primitive.

TESTS:

We test creating some text:

```
sage: text("I like Fibonacci",(3,5))
```

get_minmax_data()

Returns a dictionary with the bounding box data. Notice that, for text, the box is just the location itself.

EXAMPLES:

```
sage: T = text("Where am I?",(1,1))
sage: t=T[0]
sage: t.get_minmax_data()['ymin']
1.0
sage: t.get_minmax_data()['ymax']
1.0
```

plot3d(***kwds*)

Plots 2D text in 3D.

EXAMPLES:

```
sage: T = text("ABC",(1,1))
sage: t = T[0]
sage: s=t.plot3d()
sage: s.jmol_repr(s.testing_render_params())[0][2]
'label "ABC"'
sage: s._trans
(1.0, 1.0, 0)
```

sage.plot.text.**text**(*string*, *xy*, *clip=False*, *vertical_alignment='center'*, *rgbcolor=(0, 0, 1)*, *fontsize=10*, *horizontal_alignment='center'*, *axis_coords=False*, ***options*)

Returns a 2D text graphics object at the point (x, y).

Type `text.options` for a dictionary of options for 2D text.

2D OPTIONS:

- `fontsize` - How big the text is
- `rgbcolor` - The color as an RGB tuple

- `hue` - The color given as a hue
- `rotation` - How to rotate the text: angle in degrees, vertical, horizontal
- `vertical_alignment` - How to align vertically: top, center, bottom
- `horizontal_alignment` - How to align horizontally: left, center, right
- `axis_coords` - (default: False) if True, use axis coordinates, so that (0,0) is the lower left and (1,1) upper right, regardless of the x and y range of plotted values.

EXAMPLES:

```
sage: text("Sage is really neat!!",(2,12))
```

The same text in larger font and colored red:

```
sage: text("Sage is really neat!!",(2,12),fontsize=20,rgbcolor=(1,0,0))
```

Same text but guaranteed to be in the lower left no matter what:

```
sage: text("Sage is really neat!!",(0,0), axis_coords=True, horizontal_alignment='left')
```

Same text rotated around the left, bottom corner of the text:

```
sage: text("Sage is really neat!!",(0,0), rotation=45.0, horizontal_alignment='left', vertical_a
```

Same text oriented vertically:

```
sage: text("Sage is really neat!!",(0,0), rotation="vertical")
```

You can also align text differently:

```
sage: t1 = text("Hello",(1,1), vertical_alignment="top")
sage: t2 = text("World", (1,0.5), horizontal_alignment="left")
sage: t1 + t2   # render the sum
```

You can save text as part of PDF output:

```
sage: text("sage", (0,0), rgbcolor=(0,0,0)).save(os.path.join(SAGE_TMP, 'a.pdf'))
```

Text must be 2D (use the text3d command for 3D text):

```
sage: t = text("hi",(1,2,3))
Traceback (most recent call last):
...
ValueError: use text3d instead for text in 3d
sage: t = text3d("hi",(1,2,3))
```

Extra options will get passed on to show(), as long as they are valid:

```
sage: text("MATH IS AWESOME", (0, 0), fontsize=40, axes=False)
sage: text("MATH IS AWESOME", (0, 0), fontsize=40).show(axes=False) # These are equivalent
```

CHAPTER

TWENTYFOUR

COLORS

This module defines a `Color` object and helper functions (see, e.g., `hue()`, `rainbow()`), as well as a set of `colors` and `colormaps` to use with `Graphics` objects in Sage.

For a list of pre-defined colors in Sage, evaluate:

```
sage: sorted(colors)
['aliceblue', 'antiquewhite', 'aqua', 'aquamarine', 'automatic', ...]
```

Apart from 'automatic' which just an alias for 'lightblue', this list comprises the "official" W3C CSS3 / SVG colors.

For a list of color maps in Sage, evaluate:

```
sage: sorted(colormaps)
['Accent', 'Accent_r', 'Blues', 'Blues_r', 'BrBG', 'BrBG_r', ...]
```

These are imported from matplotlib's cm module.

class sage.plot.colors.**Color**(*r='#0000ff'*, *g=None*, *b=None*, *space='rgb'*)

Bases: `object`

An Red-Green-Blue (RGB) color model color object. For most consumer-grade devices (e.g., CRTs, LCDs, and printers), as well as internet applications, this is a point in the sRGB absolute color space. The Hue-Saturation-Lightness (HSL), Hue-Lightness-Saturation (HLS), and Hue-Saturation-Value (HSV) spaces are useful alternate representations, or coordinate transformations, of this space. Coordinates in all of these spaces are floating point values in the interval [0.0, 1.0].

Note: All instantiations of `Color` are converted to an internal RGB floating point 3-tuple. This is likely to degrade precision.

INPUT:

- `r,g,b` - either a triple of floats between 0 and 1, OR `r` - a color name string or HTML color hex string
- `space` - a string (default: 'rgb'); the coordinate system (other choices are 'hsl', 'hls', and 'hsv') in which to interpret a triple of floats

EXAMPLES:

```
sage: Color('purple')
RGB color (0.5019607843137255, 0.0, 0.5019607843137255)
sage: Color('#8000ff')
RGB color (0.5019607843137255, 0.0, 1.0)
sage: Color(0.5,0,1)
RGB color (0.5, 0.0, 1.0)
sage: Color(0.5, 1.0, 1, space='hsv')
```

```
RGB color (0.0, 1.0, 1.0)
sage: Color(0.25, 0.5, 0.5, space='hls')
RGB color (0.5000000000000001, 0.75, 0.25)
sage: Color(1, 0, 1/3, space='hsl')
RGB color (0.3333333333333333, 0.3333333333333333, 0.3333333333333333)
sage: from sage.plot.colors import chocolate
sage: Color(chocolate)
RGB color (0.8235294117647058, 0.4117647058823529, 0.11764705882352941)
```

blend(*color*, *fraction=0.5*)

Return a color blended with the given `color` by a given `fraction`. The algorithm interpolates linearly between the colors' corresponding R, G, and B coordinates.

INPUT:

- `color` - a `Color` instance or float-convertible 3-tuple/list; the color with which to blend this color
- `fraction` - a float-convertible number; the fraction of `color` to blend with this color

OUTPUT:

- a **new** `Color` instance

EXAMPLES:

```
sage: from sage.plot.colors import red, blue, lime
sage: red.blend(blue)
RGB color (0.5, 0.0, 0.5)
sage: red.blend(blue, fraction=0.0)
RGB color (1.0, 0.0, 0.0)
sage: red.blend(blue, fraction=1.0)
RGB color (0.0, 0.0, 1.0)
sage: lime.blend((0.3, 0.5, 0.7))
RGB color (0.15, 0.75, 0.35)
sage: blue.blend(blue)
RGB color (0.0, 0.0, 1.0)
sage: red.blend(lime, fraction=0.3)
RGB color (0.7, 0.3, 0.0)
sage: blue.blend((0.0, 0.9, 0.2), fraction=0.2)
RGB color (0.0, 0.18000000000000002, 0.8400000000000001)
sage: red.blend(0.2)
Traceback (most recent call last):
...
TypeError: 0.200000000000000 must be a Color or float-convertible 3-tuple/list
```

darker(*fraction=0.3333333333333333*)

Return a darker "shade" of this RGB color by `blend()`-ing it with black. This is **not** an inverse of `lighter()`.

INPUT:

- `fraction` - a float (default: 1.0/3.0); blending fraction to apply

OUTPUT:

- a new instance of `Color`

EXAMPLES:

```
sage: from sage.plot.colors import black
sage: vector(black.darker().rgb()) == vector(black.rgb())
True
```

```
sage: Color(0.4, 0.6, 0.8).darker(0.1)
RGB color (0.36000000000000004, 0.54, 0.7200000000000001)
sage: Color(.1,.2,.3,space='hsl').darker()
RGB color (0.24000000000000002, 0.20800000000000002, 0.16)
```

hls()

Return the Hue-Lightness-Saturation (HLS) coordinates of this color.

OUTPUT:

- a 3-tuple of floats

EXAMPLES:

```
sage: Color(0.3, 0.5, 0.7, space='hls').hls()
(0.30000000000000004, 0.5, 0.7)
sage: Color(0.3, 0.5, 0.7, space='hsl').hls()
(0.30000000000000004, 0.7, 0.5000000000000001)
sage: Color('#aabbcc').hls()
(0.5833333333333334, 0.7333333333333334, 0.25000000000000017)
sage: from sage.plot.colors import orchid
sage: orchid.hls()
(0.8396226415094339, 0.6470588235294117, 0.5888888888888889)
```

hsl()

Return the Hue-Saturation-Lightness (HSL) coordinates of this color.

OUTPUT:

- a 3-tuple of floats

EXAMPLES:

```
sage: Color(1,0,0).hsl()
(0.0, 1.0, 0.5)
sage: from sage.plot.colors import orchid
sage: orchid.hsl()
(0.8396226415094339, 0.5888888888888889, 0.6470588235294117)
sage: Color('#aabbcc').hsl()
(0.5833333333333334, 0.25000000000000017, 0.7333333333333334)
```

hsv()

Return the Hue-Saturation-Value (HSV) coordinates of this color.

OUTPUT:

- a 3-tuple of floats

EXAMPLES:

```
sage: from sage.plot.colors import red
sage: red.hsv()
(0.0, 1.0, 1.0)
sage: Color(1,1,1).hsv()
(0.0, 0.0, 1.0)
sage: Color('gray').hsv()
(0.0, 0.0, 0.5019607843137255)
```

html_color()

Return a HTML hex representation for this color.

OUTPUT:

•a string of length 7.

EXAMPLES:

```
sage: Color('yellow').html_color()
'#ffff00'
sage: Color('#fedcba').html_color()
'#fedcba'
sage: Color(0.0, 1.0, 0.0).html_color()
'#00ff00'
sage: from sage.plot.colors import honeydew
sage: honeydew.html_color()
'#f0fff0'
```

lighter (*fraction=0.3333333333333333*)

Return a lighter "shade" of this RGB color by `blend()`-ing it with white. This is **not** an inverse of `darker()`.

INPUT:

•`fraction` - a float (default: 1.0/3.0); blending fraction to apply

OUTPUT:

•a **new** instance of `Color`

EXAMPLES:

```
sage: from sage.plot.colors import khaki
sage: khaki.lighter()
RGB color (0.9607843137254903, 0.934640522875817, 0.6993464052287582)
sage: Color('white').lighter().darker()
RGB color (0.6666666666666667, 0.6666666666666667, 0.6666666666666667)
sage: Color('#abcdef').lighter(1/4)
RGB color (0.7529411764705882, 0.8529411764705883, 0.9529411764705882)
sage: Color(1, 0, 8/9, space='hsv').lighter()
RGB color (0.925925925925926, 0.925925925925926, 0.925925925925926)
```

rgb()

Return the underlying Red-Green-Blue (RGB) coordinates of this color.

OUTPUT:

•a 3-tuple of floats

EXAMPLES:

```
sage: Color(0.3, 0.5, 0.7).rgb()
(0.3, 0.5, 0.7)
sage: Color('#8000ff').rgb()
(0.5019607843137255, 0.0, 1.0)
sage: from sage.plot.colors import orange
sage: orange.rgb()
(1.0, 0.6470588235294118, 0.0)
sage: Color('magenta').rgb()
(1.0, 0.0, 1.0)
sage: Color(1, 0.7, 0.9, space='hsv').rgb()
(0.9, 0.2700000000000001, 0.2700000000000001)
```

class sage.plot.colors.**Colormaps**

Bases: _abcoll.MutableMapping

A dict-like collection of lazily-loaded matplotlib color maps. For a list of map names, evaluate:

```
sage: sorted(colormaps)
['Accent', 'Accent_r', 'Blues', 'Blues_r', ...]
```

load_maps()

If it's necessary, loads matplotlib's color maps and adds them to the collection.

EXAMPLES:

```
sage: from sage.plot.colors import Colormaps
sage: maps = Colormaps()
sage: len(maps.maps)
0
sage: maps.load_maps()
sage: len(maps.maps)>130
True
```

class sage.plot.colors.**ColorsDict**

Bases: dict

A dict-like collection of colors, accessible via key or attribute. For a list of color names, evaluate:

```
sage: sorted(colors)
['aliceblue', 'antiquewhite', 'aqua', 'aquamarine', ...]
```

sage.plot.colors.**float_to_html**(*r*, *g*, *b*)

Converts a Red-Green-Blue (RGB) color tuple to a HTML hex color. Each input value should be in the interval [0.0, 1.0]; otherwise, the values are first reduced modulo one (see mod_one()).

INPUT:

- r - a number; the RGB color's "red" intensity
- g - a number; the RGB color's "green" intensity
- b - a number; the RGB color's "blue" intensity

OUTPUT:

- a string of length 7, starting with '#'

EXAMPLES:

```
sage: from sage.plot.colors import float_to_html
sage: float_to_html(1.,1.,0.)
'#ffff00'
sage: float_to_html(.03,.06,.02)
'#070f05'
sage: float_to_html(*Color('brown').rgb())
'#a52a2a'
sage: float_to_html((0.2, 0.6, 0.8))
Traceback (most recent call last):
...
TypeError: float_to_html() takes exactly 3 arguments (1 given)
```

sage.plot.colors.**get_cmap**(*cmap*)

Returns a color map (actually, a matplotlib Colormap object), given its name or a [mixed] list/tuple of RGB list/tuples and color names. For a list of map names, evaluate:

```
sage: sorted(colormaps)
['Accent', 'Accent_r', 'Blues', 'Blues_r', ...]
```

See rgbcolor() for valid list/tuple element formats.

INPUT:

- cmap - a string, list, tuple, or matplotlib.colors.Colormap; a string must be a valid color map name

OUTPUT:

- a matplotlib.colors.Colormap instance

EXAMPLES:

```
sage: from sage.plot.colors import get_cmap
sage: get_cmap('jet')
<matplotlib.colors.LinearSegmentedColormap object at 0x...>
sage: get_cmap(u'jet')
<matplotlib.colors.LinearSegmentedColormap object at 0x...>
sage: get_cmap([(0,0,0), (0.5,0.5,0.5), (1,1,1)])
<matplotlib.colors.ListedColormap object at 0x...>
sage: get_cmap(['green', 'lightblue', 'blue'])
<matplotlib.colors.ListedColormap object at 0x...>
sage: get_cmap(((0.5, 0.3, 0.2), [1.0, 0.0, 0.5], 'purple', Color(0.5,0.5,1, space='hsv')))
<matplotlib.colors.ListedColormap object at 0x...>
sage: get_cmap('jolies')
Traceback (most recent call last):
...
RuntimeError: Color map jolies not known (type import matplotlib.cm; matplotlib.cm.datad.keys()
sage: get_cmap('mpl')
Traceback (most recent call last):
...
RuntimeError: Color map mpl not known (type import matplotlib.cm; matplotlib.cm.datad.keys() for
```

sage.plot.colors.**html_to_float**(*c*)

Convert a HTML hex color to a Red-Green-Blue (RGB) tuple.

INPUT:

- c - a string; a valid HTML hex color

OUTPUT:

- a RGB 3-tuple of floats in the interval [0.0, 1.0]

EXAMPLES:

```
sage: from sage.plot.colors import html_to_float
sage: html_to_float('#fff')
(1.0, 1.0, 1.0)
sage: html_to_float('#abcdef')
(0.6705882352941176, 0.803921568627451, 0.9372549019607843)
sage: html_to_float('#123xyz')
Traceback (most recent call last):
...
ValueError: invalid literal for int() with base 16: '3x'
```

sage.plot.colors.**hue**(*h*, *s=1*, *v=1*)

Convert a Hue-Saturation-Value (HSV) color tuple to a valid Red-Green-Blue (RGB) tuple. All three inputs should lie in the interval [0.0, 1.0]; otherwise, they are reduced modulo 1 (see mod_one()). In particular h=0 and h=1 yield red, with the intermediate hues orange, yellow, green, cyan, blue, and violet as h increases.

This function makes it easy to sample a broad range of colors for graphics:

```
sage: p = Graphics()
sage: for phi in xsrange(0, 2 * pi, 1 / pi):
```

```
...       p += plot(sin(x + phi), (x, -7, 7), rgbcolor = hue(phi))
sage: p
```

INPUT:

- h - a number; the color's hue
- s - a number (default: 1); the color's saturation
- v - a number (default: 1); the color's value

OUTPUT:

- a RGB 3-tuple of floats in the interval [0.0, 1.0]

EXAMPLES:

```
sage: hue(0.6)
(0.0, 0.40000000000000036, 1.0)
sage: from sage.plot.colors import royalblue
sage: royalblue
RGB color (0.2549019607843137, 0.4117647058823529, 0.8823529411764706)
sage: hue(*royalblue.hsv())
(0.2549019607843137, 0.4117647058823529, 0.8823529411764706)
sage: hue(.5, .5, .5)
(0.25, 0.5, 0.5)
```

Note: The HSV to RGB coordinate transformation itself is given in the source code for the Python library's colorsys module:

```
sage: from colorsys import hsv_to_rgb    # not tested
sage: hsv_to_rgb??                       # not tested
```

---

sage.plot.colors.**mod_one**(*x*)

Reduce a number modulo 1.

INPUT:

- x - an instance of Integer, int, RealNumber, etc.; the number to reduce

OUTPUT:

- a float

EXAMPLES:

```
sage: from sage.plot.colors import mod_one
sage: mod_one(1)
1.0
sage: mod_one(7.0)
0.0
sage: mod_one(-11/7)
0.4285714285714286
sage: mod_one(pi) + mod_one(-pi)
1.0
```

sage.plot.colors.**rainbow**(*n*, *format='hex'*)

Returns a list of colors sampled at equal intervals over the spectrum, from Hue-Saturation-Value (HSV) coordinates (0, 1, 1) to (1, 1, 1). This range is red at the extremes, but it covers orange, yellow, green, cyan, blue, violet, and many other hues in between. This function is particularly useful for representing vertex partitions on graphs.

INPUT:

- n - a number; the length of the list
- format - a string (default: 'hex'); the output format for each color in the list; the other choice is 'rgbtuple'

OUTPUT:

- a list of strings or RGB 3-tuples of floats in the interval [0.0, 1.0]

EXAMPLES:

```
sage: from sage.plot.colors import rainbow
sage: rainbow(7)
['#ff0000', '#ffda00', '#48ff00', '#00ff91', '#0091ff', '#4800ff', '#ff00da']
sage: rainbow(7, 'rgbtuple')
[(1.0, 0.0, 0.0), (1.0, 0.8571428571428571, 0.0), (0.2857142857142858, 1.0, 0.0), (0.0, 1.0, 0.5
```

AUTHORS:

- Robert L. Miller
- Karl-Dieter Crisman (directly use hsv_to_rgb() for hues)

sage.plot.colors.**rgbcolor**(*c*, *space='rgb'*)

Convert a color (string, tuple, list, or Color) to a mod-one reduced (see mod_one()) valid Red-Green-Blue (RGB) tuple. The returned tuple is also a valid matplotlib RGB color.

INPUT:

- c - a Color instance, string (name or HTML hex), 3-tuple, or 3-list; the color to convert
- space - a string (default: 'rgb'); the color space coordinate system (other choices are 'hsl', 'hls', and 'hsv') in which to interpret a 3-tuple or 3-list

OUTPUT:

- a RGB 3-tuple of floats in the interval [0.0, 1.0]

EXAMPLES:

```
sage: from sage.plot.colors import rgbcolor
sage: rgbcolor(Color(0.25, 0.4, 0.9))
(0.25, 0.4, 0.9)
sage: rgbcolor('purple')
(0.5019607843137255, 0.0, 0.5019607843137255)
sage: rgbcolor(u'purple')
(0.5019607843137255, 0.0, 0.5019607843137255)
sage: rgbcolor('#fa0')
(1.0, 0.6666666666666666, 0.0)
sage: rgbcolor(u'#fa0')
(1.0, 0.6666666666666666, 0.0)
sage: rgbcolor('#ffffff')
(1.0, 1.0, 1.0)
sage: rgbcolor(u'#ffffff')
(1.0, 1.0, 1.0)
sage: rgbcolor((1,1/2,1/3))
(1.0, 0.5, 0.3333333333333333)
sage: rgbcolor([1,1/2,1/3])
(1.0, 0.5, 0.3333333333333333)
sage: rgbcolor((1,1,1), space='hsv')
(1.0, 0.0, 0.0)
sage: rgbcolor((0.5,0.75,1), space='hls')
(0.5, 0.9999999999999999, 1.0)
```

```
sage: rgbcolor((0.5,1.0,0.75), space='hsl')
(0.5, 0.9999999999999999, 1.0)
sage: rgbcolor([1,2,255])   # WARNING -- numbers are reduced mod 1!!
(1.0, 0.0, 0.0)
sage: rgbcolor('#abcd')
Traceback (most recent call last):
...
ValueError: color hex string (= 'abcd') must have length 3 or 6
sage: rgbcolor('fff')
Traceback (most recent call last):
...
ValueError: unknown color 'fff'
sage: rgbcolor(1)
Traceback (most recent call last):
...
TypeError: '1' must be a Color, list, tuple, or string
sage: rgbcolor((0.2,0.8,1), space='grassmann')
Traceback (most recent call last):
...
ValueError: space must be one of 'rgb', 'hsv', 'hsl', 'hls'
sage: rgbcolor([0.4, 0.1])
Traceback (most recent call last):
...
ValueError: color list or tuple '[0.400000000000000, 0.100000000000000]' must have 3 entries, on
```

sage.plot.colors.**to_mpl_color**(*c*, *space='rgb'*)

Convert a color (string, tuple, list, or Color) to a mod-one reduced (see mod_one()) valid Red-Green-Blue (RGB) tuple. The returned tuple is also a valid matplotlib RGB color.

INPUT:

- c - a Color instance, string (name or HTML hex), 3-tuple, or 3-list; the color to convert
- space - a string (default: 'rgb'); the color space coordinate system (other choices are 'hsl', 'hls', and 'hsv') in which to interpret a 3-tuple or 3-list

OUTPUT:

- a RGB 3-tuple of floats in the interval [0.0, 1.0]

EXAMPLES:

```
sage: from sage.plot.colors import rgbcolor
sage: rgbcolor(Color(0.25, 0.4, 0.9))
(0.25, 0.4, 0.9)
sage: rgbcolor('purple')
(0.5019607843137255, 0.0, 0.5019607843137255)
sage: rgbcolor(u'purple')
(0.5019607843137255, 0.0, 0.5019607843137255)
sage: rgbcolor('#fa0')
(1.0, 0.6666666666666666, 0.0)
sage: rgbcolor(u'#fa0')
(1.0, 0.6666666666666666, 0.0)
sage: rgbcolor('#ffffff')
(1.0, 1.0, 1.0)
sage: rgbcolor(u'#ffffff')
(1.0, 1.0, 1.0)
sage: rgbcolor((1,1/2,1/3))
(1.0, 0.5, 0.3333333333333333)
sage: rgbcolor([1,1/2,1/3])
```

```
(1.0, 0.5, 0.3333333333333333)
sage: rgbcolor((1,1,1), space='hsv')
(1.0, 0.0, 0.0)
sage: rgbcolor((0.5,0.75,1), space='hls')
(0.5, 0.9999999999999999, 1.0)
sage: rgbcolor((0.5,1.0,0.75), space='hsl')
(0.5, 0.9999999999999999, 1.0)
sage: rgbcolor([1,2,255])   # WARNING -- numbers are reduced mod 1!!
(1.0, 0.0, 0.0)
sage: rgbcolor('#abcd')
Traceback (most recent call last):
...
ValueError: color hex string (= 'abcd') must have length 3 or 6
sage: rgbcolor('fff')
Traceback (most recent call last):
...
ValueError: unknown color 'fff'
sage: rgbcolor(1)
Traceback (most recent call last):
...
TypeError: '1' must be a Color, list, tuple, or string
sage: rgbcolor((0.2,0.8,1), space='grassmann')
Traceback (most recent call last):
...
ValueError: space must be one of 'rgb', 'hsv', 'hsl', 'hls'
sage: rgbcolor([0.4, 0.1])
Traceback (most recent call last):
...
ValueError: color list or tuple '[0.400000000000000, 0.100000000000000]' must have 3 entries, on
```

CHAPTER

TWENTYFIVE

ARCS IN HYPERBOLIC GEOMETRY

AUTHORS:

- Hartmut Monien (2011 - 08)

class sage.plot.hyperbolic_arc.**HyperbolicArc**(*A*, *B*, *options*)
Bases: sage.plot.bezier_path.BezierPath

Primitive class for hyberbolic arc type. See `hyperbolic_arc?` for information about plotting a hyperbolic arc in the complex plane.

INPUT:

- `a, b` - coordinates of the hyperbolic arc in the complex plane
- `options` - dict of valid plot options to pass to constructor

EXAMPLES:

Note that constructions should use `hyperbolic_arc`:

```
sage: from sage.plot.hyperbolic_arc import HyperbolicArc

sage: print HyperbolicArc(0, 1/2+I*sqrt(3)/2, {})
Hyperbolic arc (0.000000000000000, 0.500000000000000 + 0.866025403784439*I)
```

sage.plot.hyperbolic_arc.**hyperbolic_arc**(*a*, *b*, *rgbcolor='blue'*, *thickness=1*, *zorder=2*, *alpha=1*, *linestyle='solid'*, *fill=False*, ***options*)

Plot an arc from a to b in hyperbolic geometry in the complex upper half plane.

INPUT:

- `a, b` - complex numbers in the upper half complex plane connected bye the arc

OPTIONS:

- `alpha` - default: 1
- `thickness` - default: 1
- `rgbcolor` - default: 'blue'
- `linestyle` - (default: `'solid'`) The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.

Examples:

Show a hyperbolic arc from 0 to 1:

```
sage: hyperbolic_arc(0, 1)
```

Show a hyperbolic arc from 1/2 to i with a red thick line:

```
sage: hyperbolic_arc(1/2, I, color='red', thickness=2)
```

Show a hyperbolic arc form i to $2i$ with dashed line:

```
sage: hyperbolic_arc(I, 2*I, linestyle='dashed')
sage: hyperbolic_arc(I, 2*I, linestyle='--')
```

CHAPTER

TWENTYSIX

TRIANGLES IN HYPERBOLIC GEOMETRY

AUTHORS:

- Hartmut Monien (2011 - 08)

class sage.plot.hyperbolic_triangle.**HyperbolicTriangle**(*A*, *B*, *C*, *options*)
Bases: sage.plot.bezier_path.BezierPath

Primitive class for hyberbolic triangle type. See `hyperbolic_triangle?` for information about plotting a hyperbolic triangle in the complex plane.

INPUT:

- `a,b,c` - coordinates of the hyperbolic triangle in the upper complex plane
- `options` - dict of valid plot options to pass to constructor

EXAMPLES:

Note that constructions should use `hyperbolic_triangle`:

```
sage: from sage.plot.hyperbolic_triangle import HyperbolicTriangle
sage: print HyperbolicTriangle(0, 1/2, I, {})
Hyperbolic triangle (0.000000000000000, 0.500000000000000, 1.00000000000000*I)
```

sage.plot.hyperbolic_triangle.**hyperbolic_triangle**(*a*, *b*, *c*, *rgbcolor='blue'*, *thickness=1*, *zorder=2*, *alpha=1*, *linestyle='solid'*, *fill=False*, ***options*)

Return a hyperbolic triangle in the complex hyperbolic plane with points (a, b, c). Type `?hyperbolic_triangle` to see all options.

INPUT:

- `a, b, c` - complex numbers in the upper half complex plane

OPTIONS:

- `alpha` - default: 1
- `fill` - default: False
- `thickness` - default: 1
- `rgbcolor` - default: 'blue'
- `linestyle` - (default: `'solid'`) The style of the line, which is one of `'dashed'`, `'dotted'`, `'solid'`, `'dashdot'`, or `'--'`, `':'`, `'-'`, `'-.'`, respectively.

EXAMPLES:

Show a hyperbolic triangle with coordinates 0, $1/2 + i\sqrt{3}/2$ and $-1/2 + i\sqrt{3}/2$:

```
sage: hyperbolic_triangle(0, -1/2+I*sqrt(3)/2, 1/2+I*sqrt(3)/2)
```

A hyperbolic triangle with coordinates 0, 1 and 2+i and a dashed line:

```
sage: hyperbolic_triangle(0, 1, 2+i, fill=true, rgbcolor='red', linestyle='--')
```

CHAPTER

TWENTYSEVEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[IM] http://www.imagemagick.org

[FF] http://www.ffmpeg.org

PYTHON MODULE INDEX

g

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

V

X

Y