# CONTENTS

## 1 General

1.1 2D Plotting .................................................. 1
1.2 Text in plots .................................................. 143
1.3 Colors ....................................................... 155
1.4 Animated plots .............................................. 165

## 2 Function and Data Plots

2.1 Complex Plots ............................................... 177
2.2 Contour Plots ............................................... 185
2.3 Density Plots ................................................. 252
2.4 Plotting fields .............................................. 260
2.5 Streamline Plots ............................................ 271
2.6 Scatter Plots ............................................... 278
2.7 Step function plots ........................................ 280
2.8 Histograms .................................................. 280
2.9 Bar Charts ................................................... 283

## 3 Plots of Other Mathematical Objects

3.1 Graph Plotting ............................................... 285
3.2 Matrix Plots .................................................. 316

## 4 Basic Shapes

4.1 Arcs of circles and ellipses .................................. 323
4.2 Arrows ...................................................... 325
4.3 Bezier Paths .................................................. 338
4.4 Circles ....................................................... 344
4.5 Disks ........................................................ 356
4.6 Ellipses ........................................................ 358
4.7 Line Plots ...................................................... 360
4.8 Points ........................................................ 364
4.9 Polygons ...................................................... 369
4.10 Arcs in hyperbolic geometry ................................ 385
4.11 Polygons and triangles in hyperbolic geometry ............... 387
4.12 Regular polygons in the upper half model for hyperbolic plane 391

## 5 Infrastructure and Low-Level Functions

5.1 Graphics objects ............................................. 397
5.2 Graphics arrays and insets .................................. 423
5.3 Plotting primitives .......................................... 444
5.4 Plotting utilities ............................................. 446
1.1 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()`
• `multi_graphics()`

• 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))
```

Graphics object consisting of 1 graphics primitive

We draw a circle and a curve:

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

Graphics object consisting of 2 graphics primitives

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. Both will be measured in inches:

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

However, do not make the `figsize` too big (e.g. one dimension greater than 327 or both in the mid-200s) as this will lead to errors or crashes. See `show()` for full details.

Note that the axes will not cross if the data is not on both sides of both axes, even if it is quite close:
1.1. 2D Plotting
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
sage: plot(x^2, (x, 1, 10))
Graphics object consisting of 1 graphics primitive
```

```sage
sage: plot(x^2, (x, 300, 500)) # scientific notation on y-axis
Graphics object consisting of 1 graphics primitive
```

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

```sage
sage: plot(x^2, (x, 300, 500), ticks=[100, 50000])
Graphics object consisting of 1 graphics primitive
```

To change the ticks on one axis only, use the following notation:

```sage
sage: plot(x^2, (x, 300, 500), ticks=[None, 50000])
Graphics object consisting of 1 graphics primitive
```

You can even have custom tick labels along with custom positioning.
1.1. 2D Plotting
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
```

We combine together different graphics objects using “+”:

```
sage: H = G + P + Q
```

(continues on next page)
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
Graphics object consisting of 3 graphics primitives

A plot

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: tt = text('A Bulb', (1.5, 0.25))
sage: tx = text('x axis', (1.5,-0.2))
sage: ty = text('y axis', (0.4,0.9))
sage: g = p + tt + tx + ty
sage: g.show(xmin=-1.5, xmax=2, ymin=-1, ymax=1)
```

We can add a graphics object to another one as an inset:

```
sage: g1 = plot(x^2*sin(1/x), (x, -2, 2), axes_labels=['$x$', '$y$'])
sage: g2 = plot(x^2*sin(1/x), (x, -0.3, 0.3), axes_labels=['$x$', '$y$'],
             frame=True)
```

(continues on next page)
We can add a title to a graph:

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

Graphics object consisting of 1 graphics primitive

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))
```

Graphics object consisting of 1 graphics primitive

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

```sage
i = CDF.0
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

Graphics object consisting of 2 graphics primitives
A plot of $x^2$
2.0
 1.5
 1.0
 0.5
 0.5 1.0 1.5 2.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
Plot of $x^2$
Note: Not all functions in Sage are symbolic. When plotting non-symbolic functions they should be wrapped in `lambda`:

```
sage: plot(lambda x:fibonacci(round(x)), (x,1,10))
```

Graphics object consisting of 1 graphics primitive

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') + \
```

(continues on next page)
1.1. 2D Plotting
PYX EXAMPLES: These are some examples of plots similar to some of the plots in the PyX (http://pyx.sourceforge.net) documentation:

Symbollline:

```python
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, \rightarrow 3))
```

Cycliclink:

```python
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))
```

Pi Axis:

```python
sage: g1 = plot(sin(x), 0, 2*pi)
sage: g2 = plot(cos(x), 0, 2*pi, linestyle="--")
```
1.1. 2D Plotting
Chapter 1. General
An illustration of integration:

\[ \int_{a}^{b} f(x) \, dx \]

NUMERICAL PLOTTING:

Sage includes Matplotlib, which provides 2D plotting with an interface that is a likely very familiar to people doing numerical computation. You can use `plt.clf()` to clear the current image frame and `plt.close()` to close it. For example,

```python
sage: import pylab as plt
sage: t = plt.arange(0.0, 2.0, 0.01)
sage: s = sin(2*pi*t)
```

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

```python
sage: plt.imshow([[0.0, 0.0, 0.0]])
<matplotlib.image.AxesImage object at ...>
```

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

```python
sage: reset()
```

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

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
- 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)
- Aaron Lauve (2016-07-13): reworked handling of ‘color’ when passed a list of functions; now more in-line with other CAS’s. Added list functionality to linestyle and legend_label options as well. (trac ticket #12962)
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.

```python
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

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 \(f\).

\[
\text{sage.plot.plot} \quad \text{\texttt{graphics_array}} \quad \text{(array, nrows=None, ncols=None)}
\]
Plot a list of lists (or tuples) of graphics objects on one canvas, arranged as an array.

INPUT:
• array – either a list of lists of \(\text{Graphics}\) elements or a single list of \(\text{Graphics}\) elements
• nrows, ncols – (optional) integers. If both are given then the input array is flattened and turned into an \(nrows \times ncols\) array, with blank graphics objects padded at the end, if necessary. If only one is specified, the other is chosen automatically.

OUTPUT:
• instance of \(\text{GraphicsArray}\)

EXAMPLES: Make some plots of \(\sin\) functions:

\[
\begin{align*}
\text{sage: } & f(x) = \sin(x) \\
\text{sage: } & g(x) = \sin(2\cdot x) \\
\text{sage: } & h(x) = \sin(4\cdot x) \\
\text{sage: } & p1 = \text{plot}(f, (-2\cdot \pi, 2\cdot \pi), \text{color=hue}(0.5)) \quad \# \text{long time} \\
\text{sage: } & p2 = \text{plot}(g, (-2\cdot \pi, 2\cdot \pi), \text{color=hue}(0.9)) \quad \# \text{long time} \\
\text{sage: } & p3 = \text{parametric_plot}((f, g), (0, 2\cdot \pi), \text{color=hue}(0.6)) \quad \# \text{long time} \\
\text{sage: } & p4 = \text{parametric_plot}((f, h), (0, 2\cdot \pi), \text{color=hue}(1.0)) \quad \# \text{long time}
\end{align*}
\]

Now make a graphics array out of the plots:

\[
\text{sage: } \text{ga} = \text{graphics_array}(((p1, p2), (p3, p4))) \quad \# \text{long time}
\]
Graphics Array of size 2 x 2

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

\[
\begin{align*}
\text{sage: } & \text{ga} = \text{graphics_array}(((p1, p2), (p3, p4))) \quad \# \text{long time} \\
\text{sage: } & \text{ga}.\text{show()} \quad \# \text{long time; same output as above}
\end{align*}
\]

Here we give only one row:

\[
\begin{align*}
\text{sage: } & p1 = \text{plot}(\sin,(-4,4)) \\
\text{sage: } & p2 = \text{plot}(\cos,(-4,4)) \\
\text{sage: } & \text{ga} = \text{graphics_array}([p1, p2]) \\
\text{sage: } & \text{ga} \\
\end{align*}
\]
Graphics Array of size 1 x 2

\[
\text{sage: } \text{ga}.\text{show()}
\]

It is possible to use \texttt{figsize} to change the size of the plot as a whole:

\[
\begin{align*}
\text{sage: } & L = [\text{plot}(\sin(k\cdot x), (x,-\pi,\pi)) \quad \text{for } k \quad \text{in } \{1..3\}] \\
\text{sage: } & \text{ga} = \text{graphics_array}(L) \\
\text{sage: } & \text{ga}.\text{show}(\text{figsize}=[5,3]) \quad \# \text{smallish and compact} \\
\text{sage: } & \text{ga}.\text{show}(\text{figsize}=[5,7]) \quad \# \text{tall and thin; long time} \\
\text{sage: } & \text{ga}.\text{show}(\text{figsize}=4) \quad \# \text{width=4 inches, height fixed from default aspect ratio}
\end{align*}
\]

Specifying only the number of rows or the number of columns computes the other dimension automatically:
The options like fontsize, scale or frame passed to individual plots are preserved:

```
sage: p1 = plot(sin(x^2), (x, 0, 6),
....:     axes_labels=[r'$\theta$', r'$\sin(\theta^2)$'], fontsize=16)
sage: p2 = plot(x^3, (x, 1, 100), axes_labels=[r'$x$', r'$y$'],
....:     scale='semilogy', frame=True, gridlines='minor')
sage: ga = graphics_array([p1, p2])
sage: ga.show()
```

See also: `GraphicsArray` for more examples

`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.
\[ \sin(\theta^2) \]
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.

For other keyword options that the `list_plot` function can take, refer to `plot()`.

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

**EXAMPLES:**

```sage
sage: list_plot([i^2 for i in range(5)])  # long time
Graphics object consisting of 1 graphics primitive
```

Here are a bunch of random red points:

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

You can provide a numpy array.
sage: import numpy
sage: list_plot(numpy.arange(10))
Graphics object consisting of 1 graphics primitive

Plot a list of complex numbers:

sage: list_plot([1, I, pi + I/2, CC(.25, .25)])
Graphics object consisting of 1 graphics primitive

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)])
Graphics object consisting of 1 graphics primitive

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:
1.1. 2D Plotting

```
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(list(zip(x_coords, y_coords)))
Graphics object consisting of 1 graphics primitive
```

```
1.0
 0.5
 0.5 1.0
1.0
0.5
0.5
1.0
```

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 plot two lists 'x' and 'y' against each other, use 'list_plot(list(zip(x,y)))'.

```
Dictionaries with numeric keys and values can be plotted:

```
```
Graphics object consisting of 1 graphics primitive
```

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
Graphics object consisting of 1 graphics primitive
```

1.1. 2D Plotting
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). Further, matplotlib will display a user warning.

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

We use \texttt{list_plot\_loglog()} and plot in a different base.
We can also change the scale of the axes in the graphics just before displaying:

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

```python
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.

**INPUT:**

- `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:**

```python
sage: yl = [5**k for k in range(10)]; xl = [2**k for k in range(10)]
sage: list_plot_loglog(list(zip(xl, yl))) # long time # plot in loglog scale with base 10
Graphics object consisting of 1 graphics primitive
```

```python
sage: list_plot_loglog(list(zip(xl, yl)), base=2.1) # long time # with base 2.1
Graphics object consisting of 1 graphics primitive
```
sage: list_plot_loglog(list(zip(xl, yl)), base=(2,5))  # long time
Graphics object consisting of 1 graphics primitive

**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)
Graphics object consisting of 1 graphics primitive
```

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

```
sage: list_plot_loglog(list(zip(range(1,len(yl)), yl[1:])))
Graphics object consisting of 1 graphics primitive
```

**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.

**INPUT:**

- `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(yl)
# plot empty due to (0,1)
Graphics object consisting of 1 graphics primitive
```

```
sage: list_plot_semilogx(list(zip(range(1, len(yl)), yl[1:])))
Graphics object consisting of 1 graphics primitive
```

**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 empty due to (0,1)
Graphics object consisting of 1 graphics primitive
```

We remove `(0,1)` to fix this:

```
sage: list_plot_semilogx(list(zip(range(1, len(yl)), yl[1:])))
Graphics object consisting of 1 graphics primitive
```

```
sage: list_plot_semilogx([(1,2),(3,4),(3,-1),(25,3)], base=2)  # with base 2
Graphics object consisting of 1 graphics primitive
```

**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.

**INPUT:**

- `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:**
Warning: If \texttt{plotjoined} 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). Further, matplotlib will display a user warning.

\begin{Verbatim}
sage: xl = [2**k for k in range(12)]; yl = range(len(xl))
sage: list_plot_semilogy(list(zip(xl,yl))) # plot empty due to (1,0)
doctest:warning
...
Graphics object consisting of 1 graphics primitive
\end{Verbatim}

We remove (1,0) to fix this:

\begin{Verbatim}
sage: list_plot_semilogy(list(zip(xl[1:],yl[1:])))
Graphics object consisting of 1 graphics primitive
\end{Verbatim}

\begin{Verbatim}
sage: list_plot_semilogy([2, 4, 6, 8, 16, 31], base=2) # with base 2
Graphics object consisting of 1 graphics primitive
\end{Verbatim}
1.1. 2D Plotting
Return 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:

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

```python
sage.plot.plot.multi_graphics(graphics_list)
Plot a list of graphics at specified positions on a single canvas.

If the graphics positions define a regular array, use graphics_array() instead.

INPUT:

- `graphics_list` – a list of graphics along with their positions on the canvas; each element of `graphics_list` is either
  - a pair (graphics, position), where graphics is a Graphics object and position is the 4-tuple (left, bottom, width, height) specifying the location and size of the graphics on the canvas, all quantities being in fractions of the canvas width and height
  - or a single Graphics object; its position is then assumed to occupy the whole canvas, except for some padding; this corresponds to the default position (left, bottom, width, height) = (0.125, 0.11, 0.775, 0.77)

OUTPUT:

- instance of MultiGraphics

EXAMPLES:

multi_graphics is to be used for plot arrangements that cannot be achieved with graphics_array(), for instance:

```python
sage: g1 = plot(sin(x), (x, -10, 10), frame=True)
sage: g2 = EllipticCurve([0,0,1,-1,0]).plot(color='red', thickness=2, axes_labels=['$x$', '$y$'])
..........: + text(r"$y^2 + y = x^3 - x$", (1.2, 2), color='red')
sage: g3 = matrix_plot(matrix([[1,3,5,1], [2,4,5,6], [1,3,5,7]]))
sage: G = multi_graphics([(g1, (0.125, 0.65, 0.775, 0.77)),
..........: (g2, (0.125, 0.11, 0.775, 0.77)),
..........: (g3, (0.55, 0.18, 0.4, 0.4))])
sage: G
Multigraphics with 3 elements
```

An example with a list containing a graphics object without any specified position (the graphics, here g3, occupies then the whole canvas):
1.1. 2D Plotting

\[ y^2 + y = x^3 - x \]
sage: G = multi_graphics([g3, (g1, (0.4, 0.4, 0.2, 0.2))])
sage: G
Multigraphics with 2 elements

See also:

MultiGraphics for more examples

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:

- \( \text{funcs} \) - 2 or 3-tuple of functions, or a vector of dimension 2 or 3.
- \( \text{other options} \) - 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
sage: t = var('t')
sage: parametric_plot( (cos(t), sin(t)), (t, 0, 2*pi))
Graphics object consisting of 1 graphics primitive
```

![Graph of parametric plot with circle]

```sage
sage: parametric_plot( (sin(t), sin(2*t)), (t, 0, 2*pi), color=hue(0.6) )
Graphics object consisting of 1 graphics primitive
```

```sage
sage: parametric_plot((1, t), (t, 0, 4))
Graphics object consisting of 1 graphics primitive
```

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

```sage
sage: parametric_plot((t, t^2), (t, -4, 4), fill=True)  
Graphics object consisting of 1 graphics primitive
```

A filled Hypotrochoid:

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

```sage
sage: parametric_plot( (5*cos(x), 5*sin(x), x), (x,-12, 12), plot_points=150, color="red")  
# long time
Graphics3d Object
```

1.1. 2D Plotting
sage: y=var('y')
sage: parametric_plot( (5*cos(x), x*y, cos(x*y)), (x, -4,4), (y,-4,4)) # long time
Graphics3d Object

sage: t=var('t')
sage: parametric_plot( vector((sin(t), sin(2*t))), (t, 0, 2*pi), color='green') # long time
Graphics object consisting of 1 graphics primitive

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')
Graphics object consisting of 1 graphics primitive

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.
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 `Graphics` 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 in the rendered figure. This parameter is passed directly to the `show` procedure and it could be overwritten.
- `xmax` - ending x value in the rendered figure. This parameter is passed directly to the `show` procedure and it could be overwritten.
- `ymin` - starting y value in the rendered figure. This parameter is passed directly to the `show` procedure and it could be overwritten.
- `ymax` - ending y value in the rendered figure. This parameter is passed directly to the `show` procedure and it could be overwritten.
- `detect_poles` - (Default: False) If set to True poles are detected. If set to “show” vertical asymptotes are drawn.
- `legend_label` - a (TeX) string serving as the label for `X` in the legend. If `X` is a list, then this option can be a single string, or a list or dictionary with strings as entries/values. If a dictionary, then keys are taken from `range(len(X))`. 

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

• 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 $2\pi$, 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.

• For the other keyword options that the plot function can take, refer to the method show() and the further options below.

COLOR OPTIONS:

• color - (Default: ‘blue’) One of:
  – an RGB tuple (r,g,b) with each of r,g,b between 0 and 1.
  – a color name as a string (e.g., ‘purple’).
  – an HTML color such as ‘#aaff0b’.
  – a list or dictionary of colors (valid only if $X$ is a list): if a dictionary, keys are taken from range(len(X)); the entries/values of the list/dictionary may be any of the options above.
  – ‘automatic’ – maps to default (‘blue’) if $X$ is a single Sage object; and maps to a fixed sequence of regularly spaced colors if $X$ is a list.

• legend_color - the color of the text for $X$ (or each item in $X$) in the legend. Default color is ‘black’. Options are as in color above, except that the choice ‘automatic’ maps to ‘black’ if $X$ is a single Sage object.

• fillcolor - The color of the fill for the plot of $X$ (or each item in $X$). Default color is ‘gray’ if $X$ is a single Sage object or if color is a single color. Otherwise, options are as in color above.

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

LINE OPTIONS:

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

• linestyle - (default: "-" ) The style of the line, which is one of
  – "-" or "solid"

1.1. 2D Plotting
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)

If \( X \) is a list, then linestyle may be a list (with entries taken from the strings above) or a dictionary (with keys in range(len(X)) and values taken from the strings above).

• 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), 8 (octagon)
  - "$\ldots\$" (math TeX string)
  - (numsides, style, angle) to create a custom, regular symbol
     * numsides – the number of sides
     * style – 0 (regular polygon), 1 (star shape), 2 (asterisk), 3 (circle)
     * angle – the angular rotation in degrees

• 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.)

• fillalpha - (default: 0.5) How transparent the fill is. A number between 0 and 1.

MATPLOTLIB STYLE SHEET OPTION:
• stylesheet - (Default: classic) Support for loading a full matplotlib style sheet. Any style sheet listed in matplotlib.pyplot.style.available is acceptable. If a non-existing style is provided the default classic is applied.

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
Graphics object consisting of 1 graphics primitive
```

```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
Graphics object consisting of 1 graphics primitive
```

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([x*exp(-n*x^2)/.4 for n in [1..5]], (0, 2), aspect_ratio=.8)
```

1.1. 2D Plotting
1.1. 2D Plotting
By default, color will change from one primitive to the next. This may be controlled by modifying color option:

```python
sage: g1 = plot([x*exp(-n*x^2)/.4 for n in [1..3]], (0, 2), color='blue', aspect_ratio=.8); g1
Graphics object consisting of 3 graphics primitives
sage: g2 = plot([x*exp(-n*x^2)/.4 for n in [1..3]], (0, 2), color=['red','red','green'], linestyle=['-','--','-.'], aspect_ratio=.8); g2
Graphics object consisting of 3 graphics primitives
```

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

```python
sage: a = plot([]); a  # passing an empty list returns an empty plot
(Graphics() object)
Graphics object consisting of 0 graphics primitives
sage: a += plot(x**2); a  # append another plot
Graphics object consisting of 1 graphics primitive
sage: a += plot(x**3); a  # append yet another plot
Graphics object consisting of 2 graphics primitives
```

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

```python
sage: plot(sin(1/x), (x, -1, 1))
Graphics object consisting of 1 graphics primitive
```
1.1. 2D Plotting
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:

```python
sage: plot(x^3, (x, 0, 2))  # this one has the origin
Graphics object consisting of 1 graphics primitive
```

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:

```python
sage: plot(x^2, (x, 480, 500))  # this one has no scientific notation
Graphics object consisting of 1 graphics primitive
```

```python
sage: plot(x^2, (x, 300, 500))  # this one has scientific notation on y-axis
Graphics object consisting of 1 graphics primitive
```

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

```python
sage: plot(exp(x), 0, 2, legend_label='$e^x$')
Graphics object consisting of 1 graphics primitive
```

Sage understands TeX, so these all are slightly different, and you can choose one based on your needs:
It is possible to use a different color for the text of each label:

```python
sage: p1 = plot(sin, legend_label='sin', legend_color='red')
sage: p2 = plot(cos, legend_label='cos', legend_color='green')
sage: p1 + p2
Graphics object consisting of 2 graphics primitives
```

Prior to trac ticket #19485, legends by default had a shadowless gray background. This behavior can be recovered by setting the legend options on your plot object:

```python
sage: p = plot(sin(x), legend_label=r'$\sin(x)$')
sage: p.set_legend_options(back_color=(0.9,0.9,0.9), shadow=False)
```

If \( X \) is a list of Sage objects and `legend_label` is ‘automatic’, then Sage will create labels for each function according to their internal representation:
1.1. 2D Plotting
$\sin(x)$
If `legend_label` is any single string other than 'automatic', then it is repeated for all members of `X`:

```python
sage: plot([sin(x), tan(x), -x^2], legend_label='automatic')
Graphics object consisting of 3 graphics primitives
```

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

```python
sage: plot(sin(1.0/x), (-1, 1))
Graphics object consisting of 1 graphics primitive
```

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

```python
sage: plot(exp, (1, 10), scale='semilogy') # log axis on vertical
Graphics object consisting of 1 graphics primitive
```

```python
sage: plot_semilogy(exp, (1, 10)) # same thing
Graphics object consisting of 1 graphics primitive
```

```python
sage: plot_loglog(exp, (1, 10)) # both axes are log
Graphics object consisting of 1 graphics primitive
```
trig
trig

Chapter 1. General
1.1. 2D Plotting
Chapter 1. General
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 \texttt{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 \texttt{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.

```python
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
Graphics object consisting of 2 graphics primitives
```

It is important to mention that when we draw several graphs at the same time, parameters `xmin`, `xmax`, `ymin` and `ymax` are just passed directly to the `show` procedure. In fact, these parameters would be overwritten:

```python
sage: p=plot(x^3, x, xmin=-1, xmax=1,ymin=-1, ymax=1)
sage: q=plot(exp(x), x, xmin=-2, xmax=2, ymin=0, ymax=4)
sage: (p+q).show()
```

As a workaround, we can perform the trick:

```python
sage: p1 = line([(a,b) for a,b in zip(p[0].xdata,p[0].ydata) if (b>=-1 and b<=1)])
sage: q1 = line([(a,b) for a,b in zip(q[0].xdata,q[0].ydata) if (b>=0 and b<=4)])
sage: (p1+q1).show()
```

We can also directly plot the elliptic curve:
We can change the line style as well:

```python
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:

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

The marker can be a TeX symbol as well:

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

Sage currently ignores points that cannot be evaluated

```python
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
```
1.1. 2D Plotting
1.1. 2D Plotting
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))
```

Graphics object consisting of 1 graphics primitive

```
sage: set_verbose(0)
```

Plotting the real cube root function for negative input requires avoiding the complex numbers one would usually get. The easiest way is to use `real_nth_root(x, n)`

```
sage: plot(real_nth_root(x, 3), (x, -1, 1))
```

Graphics object consisting of 1 graphics primitive

We can also get the same plot in the following way:

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

Graphics object consisting of 1 graphics primitive
1.1. 2D Plotting
A way to plot other functions without symbolic variants is to use lambda functions:

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

Graphics object consisting of 1 graphics primitive

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', fillalpha=1)
sage: p3 = plot(sin(x), -pi, pi, fill='max')
sage: p4 = plot(sin(x), -pi, pi, fill=(1-x)/3, fillcolor='blue', fillalpha=.2)
sage: graphics_array([[p1, p2], [p3, p4]]).show(frame=True, axes=False) # long
```

The basic options for filling a list of plots:
Chapter 1. General
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: [f, 2*x+2], -7,7, fill={0: [1]}, fillcolor='#ccc').show(ymin=-20, ymax=20)
```

1.1. 2D Plotting
sage: def b(n): return lambda x: bessel_J(n, x)
sage: plot([b(n) for n in [1..5]], 0, 20, fill='axis')

Graphics object consisting of 10 graphics primitives

Note that to fill between the ith and jth functions, you must use the dictionary key-value syntax i:[j]: using 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, 20, fill={i:i-1 for i in [1..4]}, color='blue', aspect_ratio=3, ymax=3)

# long time
Graphics object consisting of 9 graphics primitives

sage: plot([b(c) for c in [1..5]], 0, 20, fill={i:i-1 for i in [1..4]}, color='blue', aspect_ratio=3)

# long time
Graphics object consisting of 9 graphics primitives

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

sage: plot(sin(x^2), (x, -3, 3), title=r'Plot of $\sin(x^2)$', axes_labels=['$x$', '$y$'])

# These labels will be nicely typeset
Graphics object consisting of 1 graphics primitive

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

# These will not
Graphics object consisting of 1 graphics primitive
1.1. 2D Plotting
Plot of $\sin(x^2)$
Plot of $\sin(x^2)$
\texttt{sage}: plot(sin(x^2), (x, -3, 3), axes_labels=['x', 'y'], axes_labels_size=2.5) \\[Huge axes labels (w.r.t. the tick marks)]\\Graphics object consisting of 1 graphics primitive

\begin{center}
\includegraphics[width=\textwidth]{sage_plot}
\end{center}

This includes options for custom ticks and formatting. See documentation for \texttt{show()} for more details.

\texttt{sage}: plot(sin(pi*x), (x, -8, 8), ticks=[[-7,-3,3,7],[-1/2,0,1/2]])
Graphics object consisting of 1 graphics primitive
1.1. 2D Plotting
```sage
plot(2*x+1, (x, 0, 5), ticks=[[0, 1, e, sqrt(20)], 2], tick_formatter="latex")
```

Graphics object consisting of 1 graphics primitive

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)
```

Graphics object consisting of 1 graphics primitive

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=[["$x_1$", "$x_2$"], ["$y_1$", "$y_2$", "$y_3$")])
```

Graphics object consisting of 1 graphics primitive

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
```

Graphics object consisting of 1 graphics primitive

A example with excluded values:

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

Graphics object consisting of 11 graphics primitives
1.1. 2D Plotting

1.1. 2D Plotting
We exclude all points where \texttt{PrimePi} makes a jump:

```python
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:

```python
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
```

\texttt{exclude} and \texttt{detect_poles} can be used together:

```python
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 \texttt{exclude} keyword argument:

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

20 Chapter 1. General
sage: plot(arcsec, (x, -2, 2), exclude=[1.5])  # x=1.5 is also excluded
Graphics object consisting of 3 graphics primitives

sage: plot(arcsec(x/2), -2, 2)  # plot should be empty; no valid points
Graphics object consisting of 0 graphics primitives
sage: plot(sqrt(x^2-1), -2, 2)  # [-1, 1] is excluded automatically
Graphics object consisting of 2 graphics primitives
sage: plot(arccsc, -2, 2)  # [-1, 1] is excluded automatically
Graphics object consisting of 2 graphics primitives
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.

INPUT:

- 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
Graphics object consisting of 1 graphics primitive

```
10^0 10^1 10^2 10^3 10^4

10^1 10^2 10^3 10^4
```

sage: plot_loglog(exp, (1,10), base=2.1) # long time # with base 2.1 on both axes
Graphics object consisting of 1 graphics primitive

sage: plot_loglog(exp, (1,10), base=(2,3))
Graphics object consisting of 1 graphics primitive

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.

INPUT:

- 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
Graphics object consisting of 1 graphics primitive
```python
sage: plot_semilogx(exp, (1,10), base=2) # with base 2
Graphics object consisting of 1 graphics primitive
```

```python
sage: s = var('s')  # Samples points logarithmically so graph is smooth
sage: f = 4000000/(4000000 + 4000*s*i - s*s)
sage: plot_semilogx(20*log(abs(f), 10), (s, 1, 1e6))
Graphics object consisting of 1 graphics primitive
```

`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.

**INPUT:**

- `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:**

```python
sage: plot_semilogy(exp, (1,10)) # long time # plot in semilogy scale, base 10
Graphics object consisting of 1 graphics primitive
```

```python
sage: plot_semilogy(exp, (1,10), base=2) # long time # with base 2
Graphics object consisting of 1 graphics primitive
```
1.1. 2D Plotting
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:

- **func** - 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')
```

Graphics object consisting of 1 graphics primitive

A red figure-8:

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

Graphics object consisting of 1 graphics primitive

A green limacon of Pascal:
```python
sage: polar_plot(2 + 2*cos(x), (x, 0, 2*pi), color=hue(0.3))
Graphics object consisting of 1 graphics primitive
```

Several polar plots:

```python
sage: polar_plot([2*sin(x), 2*cos(x)], (x, 0, 2*pi))
Graphics object consisting of 2 graphics primitives
```

A filled spiral:

```python
sage: polar_plot(sqrt, 0, 2 * pi, fill=True)
Graphics object consisting of 2 graphics primitives
```

Fill the area between two functions:

```python
sage: polar_plot(cos(4*x) + 1.5, 0, 2*pi, fill=0.5 * cos(4*x) + 2.5, fillcolor='orange')
Graphics object consisting of 2 graphics primitives
```

Fill the area between several spirals:

```python
sage: polar_plot([(1.2+k*0.2)*log(x) for k in range(6)], 1, 3 * pi, fill={0: [1], ...
```
1.1. 2D Plotting
1.1. 2D Plotting
Exclude points at discontinuities:

```python
sage: polar_plot(log(floor(x)), (x, 1, 4*pi), exclude=[1..12])
Graphics object consisting of 12 graphics primitives
```

```
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:**

```python
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)
```

Graphics Array of size 3 x 4
sage: M = [[plot(sin(k*x),(x,-pi,pi)) for k in range(3)],[plot(cos(j*x),(x,-pi,pi)) for j in range(3..5)]]
sage: graphics_array(M,6,1)  # long time (up to 4s on sage.math, 2012)
Graphics Array of size 6 x 1

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.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.

1.2 Text in plots

class sage.plot.text.Text(string, point, options)
Bases: sage.plot.primitive.GraphicPrimitive
Base class for Text graphics primitive.

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"

plot3d(s, **kwds)
Plots 2D text in 3D.

EXAMPLES:

sage: T = text("ABC",(1,1))
sage: t = T[0]
sage: s = t.plot3d()
sage: s.plot3d()
I like Fibonacci
Type `text.options` for a dictionary of options for 2D text.

**2D OPTIONS:**

- **fontsize** - How big the text is. Either an integer that specifies the size in points or a string which specifies a size (one of ‘xx-small’, ‘x-small’, ‘small’, ‘medium’, ‘large’, ‘x-large’, ‘xx-large’)
- **fontstyle** - A string either ‘normal’, ‘italic’ or ‘oblique’
- **rgbcolor** - The color as an RGB tuple
- **hue** - The color given as a hue
- **alpha** - A float (0.0 transparent through 1.0 opaque)
- **background_color** - The background color
- **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
- **zorder** - The layer level in which to draw
- **clip** - (default: False) Whether to clip or not
- **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.
- **bounding_box** - A dictionary specifying a bounding box. Currently the text location.

**EXAMPLES:**

```sage
sage: text("Sage graphics are really neat because they use matplotlib!", (2,12))
Graphics object consisting of 1 graphics primitive
```

Larger font, bold, colored red and transparent text:

```sage
sage: text("I had a dream!", (2,12), alpha=0.3, fontsize='large', fontweight='bold', color='red')
Graphics object consisting of 1 graphics primitive
```

By setting `horizontal_alignment` to ‘left’ the text is guaranteed to be in the lower left no matter what:

```sage
sage: text("I got a horse and he lives in a tree", (0,0), axis_coords=True, horizontal_alignment='left')
Graphics object consisting of 1 graphics primitive
```

Various rotations:

```sage
sage: text("noitator", (0,0), rotation=45.0, horizontal_alignment='left', vertical_alignment='bottom')
Graphics object consisting of 1 graphics primitive
```

```sage
sage: text("Sage is really neat!!",(0,0), rotation="vertical")
Graphics object consisting of 1 graphics primitive
```

You can also align text differently:
Sage graphics are really neat because they use matplotlib!
I had a dream!
I got a horse and he lives in a tree
Sage is really neat!!
```python
sage: t1 = text("Hello", (1,1), vertical_alignment="top")
```
```python
sage: t2 = text("World", (1,0.5), horizontal_alignment="left")
```
```python
sage: t1 + t2  # render the sum
```
```
Graphics object consisting of 2 graphics primitives
```

You can save text as part of PDF output:
```python
sage: text("sage", (0,0), rgbcolor=(0,0,0)).save(os.path.join(SAGE_TMP, 'a.pdf'))
```

Some examples of bounding box:
```python
sage: bbox = {'boxstyle':"rarrow,pad=0.3", 'fc':"cyan", 'ec':"b", 'lw':2}
sage: text("I feel good", (1,2), bounding_box=bbox)
```
```
Graphics object consisting of 1 graphics primitive
```
```python
sage: text("So good", (0,0), bounding_box={'boxstyle':'round', 'fc':'w'})
```
```
Graphics object consisting of 1 graphics primitive
```


A text with a background color:

1.2. Text in plots
1.2. Text in plots
sage: text("So good", (-2,2), background_color='red')
Graphics object consisting of 1 graphics primitive

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)
Graphics object consisting of 1 graphics primitive
sage: text("MATH IS AWESOME", (0, 0), fontsize=40).show(axes=False) # These are equivalent
```
1.3 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:

```python
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:

```python
sage: sorted(colormaps)
[u'Accent', u'Blues', u'BrBG', ...]
```

These are imported from matplotlib’s `cm` module.

```python
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:

```python
sage: Color('purple')
RGB color (0.5019607843137255, 0.0, 0.5019607843137255)
sage: Color('#0000ff')
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, 0.0, 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:**

```python
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, 0.84000000000000001)
sage: red.blend(0.2)
Traceback (most recent call last):
  ...TypeError: 0.2000000000000000 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: 1/3); blending fraction to apply

**OUTPUT:**
- a new instance of Color

**EXAMPLES:**

```python
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, 0.72000000000000001)
sage: Color(0.5, 0.5, 0.5, space='hls').darker()
RGB color (0.44000000000000004, 0.44000000000000004, 0.44000000000000004)
```

hls()

Return the Hue-Lightness-Saturation (HLS) coordinates of this color.

**OUTPUT:**
• a 3-tuple of floats

EXAMPLES:

```python
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, 0.50000000000000001)
sage: Color('#aabbcc').hls()
(0.5833333333333334, 0.7333333333333334, 0.250000000000000017)
sage: from sage.plot.colors import orchid
sage: orchid.hls()
(0.8396226415094339, 0.6470588235294117, 0.58888888888888889)
```

**hsl()**

Return the Hue-Saturation-Lightness (HSL) coordinates of this color.

OUTPUT:

• a 3-tuple of floats

EXAMPLES:

```python
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:

```python
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:

```python
sage: Color('yellow').html_color()
'#ffff00'
sage: Color('#fedcba').html_color()
'#fedcba'
```

(continues on next page)
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: 1/3); blending fraction to apply

**OUTPUT:**

- a new instance of `Color`

**EXAMPLES:**

```python
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 (0.6666666666666666, 0.6666666666666666, 0.6666666666666666)
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 (0.9259259259259259, 0.9259259259259259, 0.9259259259259259)
```

`rgb()`

Return the underlying Red-Green-Blue (RGB) coordinates of this color.

**OUTPUT:**

- a 3-tuple of floats

**EXAMPLES:**

```python
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:** `collections.abc.MutableMapping`

A dict-like collection of lazily-loaded matplotlib color maps. For a list of map names, evaluate:

```python
sage: sorted(colormaps)
['Accent', 'Blues', ...]
```
load_maps()
If it’s necessary, loads matplotlib’s color maps and adds them to the collection.

EXAMPLES:

```python
sage: from sage.plot.colors import Colormaps
sage: maps = Colormaps()
0
sage: maps.load_maps()
len(maps.maps)>60
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:

```python
sage: sorted(colors)
["aliceblue", "antiquewhite", "aqua", "aquamarine", ...]
```

sage.plot.colors.check_color_data(cfcm)
Make sure that the arguments are in order (coloring function, colormap).
This will allow users to use both possible orders.

EXAMPLES:

```python
sage: from sage.plot.colors import check_color_data
sage: cf = lambda x,y : (x+y) % 1
sage: cm = colormaps.autumn
sage: check_color_data((cf, cm)) == (cf, cm)
True
sage: check_color_data((cm, cf)) == (cf, cm)
True
```

sage.plot.colors.float_to_html(r, g, b)
Convert 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 real number; the RGB color’s “red” intensity
• g – a real number; the RGB color’s “green” intensity
• b – a real number; the RGB color’s “blue” intensity

OUTPUT:

• a string of length 7, starting with ‘#’

EXAMPLES:

```python
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'
```

(continues on next page)
sage.plot.colors.float_to_integer(r, g, b)
Convert a Red-Green-Blue (RGB) color tuple to an integer.

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 real number; the RGB color’s “red” intensity
- g – a real number; the RGB color’s “green” intensity
- b – a real number; the RGB color’s “blue” intensity

**OUTPUT:**
- the integer \(256^2 r_{nt} + 256 g_{nt} + b_{nt}\), where \(r_{nt}\), \(g_{nt}\), and \(b_{nt}\) are obtained from \(r\), \(g\), and \(b\) by converting from the real interval \([0.0, 1.0]\) to the integer range \(0, 1, \ldots, 255\).

**EXAMPLES:**

```python
sage: from sage.plot.colors import float_to_integer
sage: float_to_integer(1., 1., 0.)
16776960
sage: float_to_integer(.03, .06, .02)
462597
sage: float_to_integer(*Color('brown').rgb())
10824234
```

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:

```python
sage: sorted(colormaps)
[u'Accent', u'Blues', ...]
```

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:**

```python
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.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
Graphics object consisting of 20 graphics primitives
```

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:

```
```
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
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((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)
```

(continues on next page)
```
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]'
```

```
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)
```
```
1.4 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, creating a sequence of PNG files. These are then assembled to various target formats using different tools. In particular, the `convert` program from ImageMagick can be used to generate an animated GIF file. FFmpeg (with the command line program `ffmpeg`) provides support for various video formats, but also an alternative method of generating animated GIFs. For browsers which support it, APNG can be used as another alternative which works without any extra dependencies.

**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 the websites of ImageMagick or FFmpeg for installation instructions.

**EXAMPLES:**

The sine function:

```python
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,.2)]
sage: a = animate(sines)
sage: a  # optional -- ImageMagick
```

(continues on next page)
Animation with 5 frames
\[
sage: \text{a.show()} \quad \# \text{optional -- ImageMagick}
\]

Animate using FFmpeg instead of ImageMagick:
\[
\begin{align*}
sage: & f = \text{tmp_file}('name=\text{'gif'}') \\
&sage: \text{a.save(filename=f, use_ffmpeg=True)} \quad \# \text{optional -- ffmpeg}
\end{align*}
\]

Animate as an APNG:
\[
sage: \text{a.apng()} \quad \# \text{long time}
\]

An animated \texttt{sage.plot.multigraphics.GraphicsArray} of rotating ellipses:
\[
\begin{align*}
sage: & E = \text{animate}((\text{graphics_array([\{ellipse((0,0),a,b,angle=t,xmin=-3,} \\
-\rightarrow xmax=3)+circle((0,0),3,\text{color='blue'} \}) \text{ for } a \text{ in range} (1,3) \text{ for } b \text{ in range} (2,4)})) \text{ for } \\
-\rightarrow t \text{ in } \text{sxrange}(0,0.1/4,0.15)) \\
sage: & \text{str(E)} \quad \# \text{animations produced from a generator do not have a known length} \\
'sage: & \text{'Animation with unknown number of frames'}' \\
sage: & E.\text{show()} \quad \# \text{optional -- ImageMagick}
\end{align*}
\]

A simple animation of a circle shooting up to the right:
\[
\begin{align*}
sage: & c = \text{animate}([\text{circle}((i,i), 1-1/(i+1), \text{hue}=i/10) \text{ for } i \text{ in } \text{srange}(0,2,0.2)], \\
-\rightarrow \ldots: & \text{xmin=0,ymin=0,xmax=2,ymax=2,figsize=[2,2]} \\
sage: & c.\text{show()} \quad \# \text{optional -- ImageMagick}
\end{align*}
\]

Animations of 3d objects:
\[
\begin{align*}
sage: & \text{var('s,t')} \\
&s,s,t) \\
sage: & \text{def sphere_and_plane}(x): \\
& \ldots: & \text{return } \text{sphere}((0,0,0),1,\text{color='red',opacity=.5})+\text{parametric_plot3d}([t,x,s], \\
-\rightarrow (s,-1,1),(t,-1,1),\text{color='green',opacity=.7}) \\
sage: & \text{sp} = \text{animate}([\text{sphere_and_plane}(x) \text{ for } x \text{ in } \text{sxrange}(-1,1,3)]) \\
sage: & \text{sp}[0] \quad \# \text{first frame} \\
& \text{Graphics3d Object} \\
sage: & \text{sp}[-1] \quad \# \text{last frame} \\
& \text{Graphics3d Object} \\
sage: & \text{sp}.\text{show()} \quad \# \text{optional -- ImageMagick}
\end{align*}
\]

If the input objects do not have a \texttt{save_image} method, then the animation object attempts to make an image by calling its internal method \texttt{sage.plot.animate.Animation.make_image()}. This is illustrated by the following example:
\[
\begin{align*}
sage: & t = \text{var('t')} \\
sage: & a = \text{animate}((\text{sin}(c*pi*t) \text{ for } c \text{ in } \text{srange}(1,2,2))) \\
sage: & a.\text{show()} \quad \# \text{optional -- ImageMagick}
\end{align*}
\]
AUTHORS:
- William Stein
- John Palmieri
- Niles Johnson (2013-12): Expand to animate more graphics objects
- Martin von Gagern (2014-12): Added APNG support
- Joshua Campbell (2020): interactive animation via Three.js viewer

REFERENCES:
- ImageMagick
- FFmpeg
- APNG
- browsers which support it

class `sage.plot.animate.APngAssembler`(out, num_frames, num_plays=0, delay=200, delay_denominator=100)

Bases: object

Builds an APNG (Animated PNG) from a sequence of PNG files. This is used by the `sage.plot.animate.Animation.apng()` method.

This code is quite simple; it does little more than copying chunks from input PNG files to the output file. There is no optimization involved. This does not depend on external programs or libraries.

INPUT:
- `out` – a file opened for binary writing to which the data will be written
- `num_frames` – the number of frames in the animation
- `num_plays` – how often to iterate, 0 means infinitely
- `delay` – numerator of the delay fraction in seconds
- `delay_denominator` – denominator of the delay in seconds

EXAMPLES:

```python
sage: from sage.plot.animate import APngAssembler
sage: def assembleAPNG():
...:     a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
...:                  xmin=0, xmax=2*pi, figsize=[2,1])
...:     pngdir = a.png()
...:     outfile = sage.misc.temporary_file.tmp_filename(ext='.png')
...:     with open(outfile, "wb") as f:
...:         apng = APngAssembler(f, len(a))
...:         for i in range(len(a)):
...:             png = os.path.join(pngdir, "{:08d}.png".format(i))
...:             apng.add_frame(png, delay=10*i + 10)
...:     return outfile
sage: assembleAPNG()  # long time
'...png'
```

`add_frame(pngfile, delay=None, delay_denominator=None)`

Adds a single frame to the APNG file.

INPUT:
• `pngfile` – file name of the PNG file with data for this frame
• `delay` – numerator of the delay fraction in seconds
• `delay_denominator` – denominator of the delay in seconds

If the delay is not specified, the default from the constructor applies.

`set_default(pngfile)`

Adds a default image for the APNG file.

This image is used as a fallback in case some application does not understand the APNG format. This method must be called prior to any calls to the `add_frame` method, if it is called at all. If it is not called, then the first frame of the animation will be the default.

**INPUT:**

• `pngfile` – file name of the PNG file with data for the default image

---

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

**Bases:** sage.misc.fast_methods.WithEqualityById, 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
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
# optional -- ImageMagick
Animation with 21 frames
sage: a[:5]  # optional -- ImageMagick
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
sage: a.show(delay=50, iterations=4)  # optional -- ImageMagick
```

An animation of drawing a parabola:

```sage
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)
```

(continues on next page)
\begin{verbatim}
sage: a.show() # optional -- ImageMagick
sage: show(L)
\end{verbatim}

\texttt{apng}(\texttt{savefile=None, show_path=False, delay=20, iterations=0})
Creates an animated PNG composed from rendering the graphics objects in self. Return the absolute path to that file.

Notice that not all web browsers are capable of displaying APNG files, though they should still present the first frame of the animation as a fallback.

The generated file is not optimized, so it may be quite large.

Input:
\begin{itemize}
\item \texttt{delay} - (default: 20) delay in hundredths of a second between frames
\item \texttt{savefile} - file that the animated gif gets saved to
\item \texttt{iterations} - integer (default: 0); number of iterations of animation. If 0, loop forever.
\item \texttt{show_path} - boolean (default: False); if True, print the path to the saved file
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
sage: a = animate([\sin(x + \text{float}(k)) \text{ for } k \text{ in } \text{srange}(0,2*\text{pi},0.7)],
....:
   xmin=0, xmax=2*\text{pi}, figsize=[2,1])
sage: dir = \text{tmp}\_\text{dir}()
sage: a.apng() # long time
sage: a.apng(savefile=dir + 'my\_animation.png', delay=35, iterations=3) # long time
sage: a.apng(savefile=dir + 'my\_animation.png', show_path=True) # long time
Animation saved to .../my\_animation.png.
\end{verbatim}

If the individual frames have different sizes, an error will be raised:
\begin{verbatim}
sage: a = animate([\text{plot}(\sin(x), (x, 0, k)) \text{ for } k \text{ in } \text{range}(1,4)],
....:
   y\text{min}=-1, y\text{max}=1, aspect\_ratio=1, figsize=[2,1])
sage: a.apng() # long time
Traceback (most recent call last):
  ...
ValueError: Chunk IHDR mismatch
\end{verbatim}

\texttt{ffmpeg}(\texttt{savefile=None, show_path=False, output_format=None, ffmpeg_options="", delay=None, iterations=0, pix_fmt='rgb24'})
Return a movie showing an animation composed from rendering the frames in self.

This method will only work if ffmpeg is installed. See \url{https://www.ffmpeg.org} for information about ffmpeg.

\textbf{INPUT:}
\begin{itemize}
\item \texttt{savefile} - file that the mpeg gets saved to.
\item \texttt{show_path} - boolean (default: False); if True, print the path to the saved file
\item \texttt{output_format} - string (default: None); format and suffix to use for the video. This may be \texttt{'mpg'}, \texttt{'mpeg'}, \texttt{'avi'}, \texttt{'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.verbose.setVerbose()` with level=1 to see additional output.

EXAMP les:

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

gif

```
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 the web sites of ImageMagick and FFMpeg for more
details. 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, ymin=-1, ymax=1, figsize=[2,1])
sage: td = tmp_dir()
sage: a.gif()
# not tested
sage: a.gif(savefile=td + 'my_animation.gif', delay=35, iterations=3) #
˓→optional -- ImageMagick
sage: with open(td + 'my_animation.gif', 'rb') as f: print(b'\x21\xf9\x04\x08\
˓→x23\x00' in f.read())
# optional -- ImageMagick
True
sage: a.gif(savefile=td + 'my_animation.gif', show_path=True) # optional -˓→ImageMagick
Animation saved to .../my_animation.gif.
sage: a.gif(savefile=td + 'my_animation_2.gif', show_path=True, use_
˓→ffmpeg=True) # optional -- ffmpeg
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.

graphics_array(ncols=3)
Return a sage.plot.multigraphics.GraphicsArray with plots of the frames of this animation, using the given number of columns. The frames must be acceptable inputs for sage.plot.
multigraphics.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, axes=False)
sage: a
# optional -- ImageMagick
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=[6,3]) # not tested

Specify different arrangement of array and save it with a given 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.save(f)

1.4. Animated plots

171


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()
sage: g
Graphics Array of size 2 x 3
```

**`interactive(**kwds)**
Create an interactive depiction of the animation.

**INPUT:**
- ***kwds** – any of the viewing options accepted by `show()` are valid as keyword arguments to this function and they will behave in the same way. Those that are animation-related and recognized by the Three.js viewer are: `animate`, `animation_controls`, `auto_play`, `delay`, and `loop`.

**OUTPUT:**
A 3D graphics object which, by default, will use the Three.js viewer.

**EXAMPLES:**
```
sage: frames = [point3d((sin(x), cos(x), x)) for x in (0, pi/16, .., 2*pi)]
sage: animate(frames).interactive(online=True)
Graphics3d Object
```

Works with frames that are 2D or 3D graphics objects or convertible to 2D or 3D graphics objects via a `plot` or `plot3d` method:
```
sage: frames = [dodecahedron(), circle(center=(0, 0), radius=1), x^2]
sage: animate(frames).interactive(online=True, delay=100)
Graphics3d Object
```

See also:
Three.js JavaScript WebGL Renderer

**`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':1, 'ymax':1}) for i in range(4)])

sage: d = B.png(); v = os.listdir(d); v.sort(); v  # long time
['00000000.png', '00000001.png', '00000002.png', '00000003.png']

sage: B.show()  # not tested
```
(continues on next page)
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()  # not tested

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:**

```python
sage: a = animate([plot(x^2 + n) for n in range(4)], ymin=0, ymax=4)
sage: d = a.png(); v = os.listdir(d); v.sort(); v  # long time
['00000000.png', '00000001.png', '00000002.png', '00000003.png']
```

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

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 ‘.html’, save an `interactive()` version of the animation to an HTML file that uses the Three.js viewer. 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. sobj and HTML files can be saved with no extra software installed.

**EXAMPLES:**

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

show (delay=None, iterations=None, **kwds)
Show this animation immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

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.
• format - (default: gif) format to use for output. Currently supported formats are: gif, ogg, webm, mp4, flash, matroska, avi, wmv, quicktime.

OUTPUT:
This method does not return anything. Use save() if you want to save the figure as an image.

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

You can also make use of the HTML5 video element in the Sage Notebook:

sage: a.show(format="ogg")  # optional -- ffmpeg
sage: a.show(format="webm")  # optional -- ffmpeg
sage: a.show(format="mp4")  # optional -- ffmpeg
sage: a.show(format="webm", iterations=1)  # optional -- ffmpeg
Other backends may support other file formats as well:

```plaintext
sage: a.show(format="flash")  # optional -- ffmpeg
sage: a.show(format="matroska")  # optional -- ffmpeg
sage: a.show(format="avi")  # optional -- ffmpeg
sage: a.show(format="wmv")  # optional -- ffmpeg
sage: a.show(format="quicktime")  # optional -- ffmpeg
```

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

```plaintext
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.

```

```
animate(frames, **kwds)
```

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

**EXAMPLES:**

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

See also `sage.plot.animate` for more examples.
2.1 Complex Plots

class sage.plot.complex_plot.ComplexPlot(rgb_data, x_range, y_range, 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.
    • x_range – A minimum and maximum x value for the plot.
    • y_range – A minimum and maximum y value for the plot.

    get_minmax_data()
    Return 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: 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, x_range, y_range, 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 x_range and y_range 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: 'cathom'), the interpolation method to use: 'bilinear', 'bicubic', 'spline16', 'spline36', 'quadric', 'gaussian', 'sinc', 'bessel', 'mitchell', 'lanczos', 'cathom', 'hermite', 'hanning', 'hamming', 'kaiser'

EXAMPLES:
Here we plot a couple of simple functions:

```python
sage: complex_plot(sqrt(x), (-5, 5), (-5, 5))
Graphics object consisting of 1 graphics primitive
```

```

sage: complex_plot(sin(x), (-5, 5), (-5, 5))
Graphics object consisting of 1 graphics primitive
```

```

sage: complex_plot(log(x), (-10, 10), (-10, 10))
Graphics object consisting of 1 graphics primitive
```

```

sage: complex_plot(exp(x), (-10, 10), (-10, 10))
Graphics object consisting of 1 graphics primitive
```

A function with some nice zeros and a pole:

```python
sage: f(z) = z^5 + z - 1 + 1/z
sage: complex_plot(f, (-3, 3), (-3, 3))
Graphics object consisting of 1 graphics primitive
```
2.1. Complex Plots
Chapter 2. Function and Data Plots
Here is the identity, useful for seeing what values map to what colors:

```
sage: complex_plot(lambda z: z, (-3, 3), (-3, 3))
```

Graphics object consisting of 1 graphics primitive

```
The Riemann Zeta function:
```

```
sage: complex_plot(zeta, (-30,30), (-30,30))
```

Graphics object consisting of 1 graphics primitive

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])
```

Graphics object consisting of 1 graphics primitive

```
sage: complex_plot(lambda z: z, (-3, 3), (-3, 3)).show(figsize=[1,1])  # These are equivalent
```

```
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.
Chapter 2. Function and Data Plots
EXAMPLES:

```python
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]])
```

```python
sage: complex_to_rgb([[0, 1j, 1000j]])
array([[0. , 0. , 0. ],
       [0.38586284, 0.77172568, 0. ],
       [0.82210588, 1. , 0.64421177]])
```

### 2.2 Contour Plots

```python
class sage.plot.contour_plot.ContourPlot(xy_data_array, xrange, yrange, options)
```

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`:

```python
sage: from sage.plot.contour_plot import ContourPlot
sage: C = ContourPlot([[1,3],[2,4]], (1,2), (2,3), options={})
```

```python
sage: C.xrange
```

```python
get_minmax_data()
```

Return a dictionary with the bounding box data.

**EXAMPLES:**

```python
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
```
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
sage: x, y = var('x,y')
sage: contour_plot(cos(x^2 + y^2), (x,-4,4), (y,-4,4))
Graphics object consisting of 1 graphics primitive
```

Here we change the ranges and add some options:

```sage
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)
Graphics object consisting of 1 graphics primitive
```

An even more complicated plot:

```sage
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)
Graphics object consisting of 1 graphics primitive
```

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
sage: x, y = var('x,y')
sage: contour_plot(y^2 + 1 - x^3 - x, (y,-pi,pi), (x,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

```sage
sage: contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

We can play with the contour levels:
2.2. Contour Plots
```python
sage: x, y = var('x, y')
sage: f(x, y) = x^2 + y^2
sage: contour_plot(f, (-2, 2), (-2, 2))
Graphics object consisting of 1 graphics primitive
```

We can change the style of the lines:

```python
sage: contour_plot(f, (-2, 2), (-2, 2), fill=False, linewidths=10)
Graphics object consisting of 1 graphics primitive
```
Chapter 2. Function and Data Plots
2.2. Contour Plots
2.2. Contour Plots
Chapter 2. Function and Data Plots
We can add labels and play with them:
Chapter 2. Function and Data Plots
sage: contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),
.....: fill=False, cmap='hsv', labels=True)
Graphics object consisting of 1 graphics primitive

sage: P=contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),
.....: fill=False, cmap='hsv',
.....: labels=True, label_fmt="%.0f",
.....: label_colors='black')
sage: P
Graphics object consisting of 1 graphics primitive

sage: P = contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),
.....: fill=False, cmap='hsv', labels=True,
.....: contours=[-4,0,4],
.....: label_fmt={-4:"low", 0:"medium", 4: "hi"},
.....: label_colors='black')
sage: P
Graphics object consisting of 1 graphics primitive

sage: P = contour_plot(y^2 + 1 - x^3 - x, (x,-pi,pi), (y,-pi,pi),
.....: fill=False, cmap='hsv', labels=True,
.....: contours=[-4,0,4],
.....: label_fmt=lambda x: "$z=%.0f$",
.....: label_colors='black', label_inline=True,
(continues on next page)
We can change the color of the labels if so desired:
2.2. Contour Plots
2.2. Contour Plots
We can add a colorbar as well:

```sage
sage: f(x, y)=x^2-y^2
sage: contour_plot(f, (x,-3,3), (y,-3,3), colorbar=True)
Graphics object consisting of 1 graphics primitive
```

```sage
sage: contour_plot(f, (x,-3,3), (y,-3,3), colorbar=True, colorbar_orientation='horizontal')
Graphics object consisting of 1 graphics primitive
```

```sage
sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[-2,-1,4], colorbar=True)
Graphics object consisting of 1 graphics primitive
```

```sage
sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[-2,-1,4], colorbar=True, colorbar_spacing='uniform')
Graphics object consisting of 1 graphics primitive
```

```sage
sage: contour_plot(f, (x,-3,3), (y,-3,3), contours=[0,2,3,6], colorbar=True, colorbar_format='%.3f')
Graphics object consisting of 1 graphics primitive
```
2.2. Contour Plots
2.2. Contour Plots
Chapter 2. Function and Data Plots
2.2. Contour Plots

215
```python
sage: contour_plot(f, (x,-3,3), (y,-3,3), labels=True,
....:     label_colors='red', contours=[0,2,3,6],
....:     colorbar=True)
Graphics object consisting of 1 graphics primitive
```

This should plot concentric circles centered at the origin:

```python
sage: x,y = var('x,y')
sage: contour_plot(x^2 + y^2 - 2, (x,-1,1), (y,-1,1))
Graphics object consisting of 1 graphics primitive
```

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

```python
sage: f(x,y) = cos(x) + sin(y)
sage: contour_plot(f, (0,pi), (0,pi), axes=True)
Graphics object consisting of 1 graphics primitive

sage: contour_plot(f, (0,pi), (0,pi)).show(axes=True) # These are equivalent
```

One can also plot over a reduced region:
2.2. Contour Plots

- A contour plot is a graphical technique for representing a 3-dimensional or multivariate dataset by plotting 2-dimensional slices of it, at a fixed level of one variable.

- The plot shows the varying values of the third variable using contour lines (also called isolines) at constant values.

- Each contour line connects points with the same value.

- The color bar on the right indicates the range of values represented by the contours.

- The axes represent the two dimensions of the data, while the third dimension is represented by the contour lines and the color bar.

- Contour plots are useful for visualizing complex data and identifying patterns or trends in the data.
2.2. Contour Plots
\texttt{sage: contour_plot(x^2 - y^2, (x,-2,2), (y,-2,2), region=x - y, plot_points=300)}

Graphics object consisting of 1 graphics primitive

Note that with \texttt{fill=False} and grayscale contours, there is the possibility of confusion between the contours and the axes, so use \texttt{fill=False} together with \texttt{axes=True} with caution:

\texttt{sage: contour_plot(f, (-pi,pi), (-pi,pi), fill=False, axes=True)}

Graphics object consisting of 1 graphics primitive

\texttt{sage.plot.contour_plot.equify(f)}

Return the equation rewritten as a symbolic function to give negative values when \texttt{True}, positive when \texttt{False}.

\textbf{EXAMPLES:}

\texttt{sage: from sage.plot.contour_plot import equify}
\texttt{sage: var(’x, y’) (x, y)}
\texttt{sage: equify(x^2 < 2)}
\texttt{x^2 - 2}
\texttt{sage: equify(x^2 > 2)}
\texttt{-x^2 + 2}
\texttt{sage: equify(x+y > 1)}
\texttt{-x*y + 1}
\texttt{sage: equify(y > 0)
2.2. Contour Plots
\begin{lstlisting}[language=python]
-sage: f = equify(lambda x, y: x > y)
-sage: f(1, 2)
1
-sage: f(2, 1)
-1

\texttt{sage.plot.contour_plot.\texttt{implicit_plot}}(f, \texttt{xrange}, \texttt{yrange}, \texttt{plot_points=150}, \texttt{contours=0},
\phantom{\texttt{plot_points=150, }}\texttt{fill=False, cmap=['blue'], **options})

\texttt{implicit_plot} takes a function of two variables, \(f(x, y)\) and plots the curve \(f(x, y) = 0\) over the specified \texttt{xrange} and \texttt{yrange} as demonstrated below.

\texttt{implicit_plot}(f, (xmin, xmax), (ymin, ymax), ...)
\texttt{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:

- \texttt{plot_points} – integer (default: 150); number of points to plot in each direction of the grid
- \texttt{fill} – boolean (default: False); if True, fill the region \(f(x, y) < 0\).
- \texttt{fillcolor} – string (default: 'blue'), the color of the region where \(f(x, y) < 0\) if \texttt{fill} = True. Colors are defined in \texttt{sage.plot.colors}; try \texttt{colors?} to see them all.
- \texttt{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.
- \texttt{linestyle} – string (default: None), the style of the line to be plotted, one of: "solid", "dashed", "dashed", or "dotted", respectively "--", "-.", "-.", or ":".
- \texttt{color} – string (default: 'blue'), the color of the plot. Colors are defined in \texttt{sage.plot.colors}; try \texttt{colors?} to see them all. If \texttt{fill} = True, then this sets only the color of the border of the plot. See \texttt{fillcolor} for setting the color of the fill region.
- \texttt{legend_label} – the label for this item in the legend
- \texttt{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 (\texttt{basex}, \texttt{basey}). \texttt{basex} sets the base of the logarithm along the horizontal axis and \texttt{basey} sets the base along the vertical axis.
- \texttt{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 (\texttt{scale, base}) or (\texttt{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 \texttt{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:
We can do the same thing, but using a callable function so we do not need to explicitly define the variables in the ranges. We also fill the inside:

```python
sage: f(x, y) = x^2 + y^2 - 2
sage: implicit_plot(f, (-3,3), (-3,3), fill=True, plot_points=500) # long time
Graphics object consisting of 2 graphics primitives
```

The same circle but with a different line width:

```python
sage: implicit_plot(f, (-3,3), (-3,3), linewidth=6)
Graphics object consisting of 1 graphics primitive
```

Again the same circle but this time with a dashdot border:

```python
sage: implicit_plot(f, (-3,3), (-3,3), linestyle='dashdot')
Graphics object consisting of 1 graphics primitive
```

The same circle with different line and fill colors:

```
2.2. Contour Plots 223```
2.2. Contour Plots
You can also plot an equation:

```sage
sage: var("x y")
(x, y)
sage: implicit_plot(x^2 + y^2 == 2, (x,-3,3), (y,-3,3))
Graphics object consisting of 1 graphics primitive
```

You can even change the color of the plot:

```sage
sage: implicit_plot(x^2 + y^2 == 2, (x,-3,3), (y,-3,3), color="red")
Graphics object consisting of 1 graphics primitive
```

The color of the fill region can be changed:

```sage
sage: implicit_plot(x**2 + y**2 == 2, (x,-3,3), (y,-3,3), fill=True, fillcolor=˓→'red')
Graphics object consisting of 2 graphics primitives
```

Here is a beautiful (and long) example which also tests that all colors work with this:
2.2. Contour Plots
We can define a level-\( n \) approximation of the boundary of the Mandelbrot set:

```python
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:

```python
sage: implicit_plot(mandel(1), (-3,3), (-3,3))
```

Graphics object consisting of 1 graphics primitive
A third-level approximation starts to get interesting:

```python
sage: implicit_plot(mandel(3), (-2,1), (-1.5,1.5))
```

Graphics object consisting of 1 graphics primitive

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.)

```python
sage: implicit_plot(mandel(7), (-0.3, 0.05), (-1.15, -0.9), plot_points=50)
```

Graphics object consisting of 1 graphics primitive

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:

```python
sage: implicit_plot(lambda x, y: x^2 + y^2 - 2, (x,-3,3), (y,-3,3),
....: fill=True, plot_points=500)  # long time
```

Graphics object consisting of 2 graphics primitives

An example of an implicit plot on ‘loglog’ scale:

```python
sage: implicit_plot(x^2 + y^2 == 200, (x,1,200), (y,1,200), scale='loglog')
```

Graphics object consisting of 1 graphics primitive

`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.
2.2. Contour Plots
Chapter 2. Function and Data Plots
region_plot(f, (xmin,xmax), (ymin,ymax), ...)

INPUT:
• f – a boolean function or a list of boolean functions 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: None), 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
• alpha – (default: 1) how transparent the fill is; a number between 0 and 1
• 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
sage: x,y = var('x,y')
sage: region_plot(cos(x^2 + y^2) <= 0, (x,-3,3), (y,-3,3))
Graphics object consisting of 1 graphics primitive
```

Here we play with the colors:

```sage
sage: region_plot(x^2 + y^3 < 2, (x,-2,2), (y,-2,2), incol='lightblue', bordercol='gray')
Graphics object consisting of 2 graphics primitives
```

An even more complicated plot, with dashed borders:

```sage
sage: region_plot(sin(x) * sin(y) >= 1/4, (x,-10,10), (y,-10,10),
....:     incol='yellow', bordercol='black',
```

(continues on next page)
2.2. Contour Plots
A disk centered at the origin:

```
sage: region_plot(x^2 + y^2 < 1, (x,-1,1), (y,-1,1))
Graphics object consisting of 1 graphics primitive
```

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))
Graphics object consisting of 1 graphics primitive
```

Since it does not look very good, let us increase `plot_points`:

```
sage: region_plot([x^2 + y^2 < 1, x < y], (x,-2,2), (y,-2,2), plot_points=400)
Graphics object consisting of 1 graphics primitive
```

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)
Graphics object consisting of 1 graphics primitive
```
2.2. Contour Plots
The first quadrant of the unit circle:

```python
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)
```

Graphics object consisting of 1 graphics primitive

Here is another plot, with a huge border:

```python
sage: region_plot(x*(x-1)*(x+1) + y^2 < 0, (x,-3,2), (y,-3,3),
     incol='lightblue', bordercol='gray', borderwidth=10,
     plot_points=50)
```

Graphics object consisting of 2 graphics primitives

If we want to keep only the region where x is positive:

```python
sage: region_plot([x*(x-1)*(x+1) + y^2 < 0, x > -1], (x,-3,2), (y,-3,3),
     incol='lightblue', plot_points=50)
```

Graphics object consisting of 1 graphics primitive

Here we have a cut circle:

```python
sage: region_plot([x^2 + y^2 < 4, x > -1], (x,-2,2), (y,-2,2),
     incol='lightblue', bordercol='gray', plot_points=200)
```

Graphics object consisting of 2 graphics primitives
2.2. Contour Plots
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))
```

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')
```

2.2. Contour Plots
2.2. Contour Plots
2.3 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:

```python
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()
{}
```

`get_minmax_data()`
Returns a dictionary with the bounding box data.

EXAMPLES:

```python
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
```

density_plot(f, xrange, yrange, plot_points=25, cmap='gray', interpolation='catrom', **options)

density_plot takes a function of two variables, \(f(x, y)\) and plots the height 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 (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.

• **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:

```python
sage: x, y = var('x, y')
sage: density_plot(sin(x) * sin(y), (x,-2,2), (y,-2,2))
```

An even more complicated plot:
This should show a "spotlight" right on the origin:

```python
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 $x, y$:

```python
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:

```python
sage: density_plot(log(x) + log(y), (x,1,10), (y,1,10), dpi=20)
```
2.3. Density Plots
2.3. Density Plots

![Density Plot Example]
2.4 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))
```

sage.plot.plot_field.plot_vector_field(f_g, xrange, yrange, plot_points=20, frame=True, **options)

plot_vector_field takes two functions of two variables xvar and yvar (for instance, if the variables are

```
sage: density_plot(log(x) + log(y), (x,1,10), (y,1,10)).show(dpi=20) # These are equivalent
```
2.4. Plotting fields
2.4. Plotting fields
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))
```

Graphics object consisting of 1 graphics primitive

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')
```

Graphics object consisting of 1 graphics primitive

Plot two orthogonal vector fields:
2.4. Plotting fields
Chapter 2. Function and Data Plots
We ignore function values that are infinite or NaN:

```python
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))
```

Graphics object consisting of 1 graphics primitive

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

```python
sage: plot_vector_field((x,y), (x,-2,2), (y,-2,2), xmax=10)
```

Graphics object consisting of 1 graphics primitive

```python
sage: plot_vector_field((x,y), (x,-2,2), (y,-2,2)).show(xmax=10) # These are equivalent
```

2.4. Plotting fields
Chapter 2. Function and Data Plots
2.4. Plotting fields
Chapter 2. Function and Data Plots
2.5 Streamline Plots

```python
class sage.plot.streamline_plot.StreamlinePlot(xpos_array, ypos_array, xvec_array, yvec_array, options):
    Bases: sage.plot.primitive.GraphicPrimitive

    Primitive class that initializes the StreamlinePlot graphics type

    get_minmax_data()
    Returns a dictionary with the bounding box data.
```

**EXAMPLES:**

```python
sage: x, y = var('x y')
sage: d = streamline_plot((.01*x, x+y), (x,10,20), (y,10,20)).get_minmax_data()
sage: d['xmin']
10.0
sage: d['ymin']
10.0
```

```python
sage.plot.streamline_plot.streamline_plot(f_g, xrange, yrange, plot_points=20, density=1.0, frame=True, **options)
Return a streamline plot in a vector field.
```

**streamline_plot** can take either one or two functions. Consider two variables $x$ and $y$.

If given two functions $(f(x, y), g(x, y))$, then this function plots streamlines in the vector field over the specified ranges with `xrange` being of $x$, denoted by `xvar` below, between $x_{min}$ and $x_{max}$, and `yrange` similarly (see below).

```python
streamline_plot((f, g), (xvar, xmin, xmax), (yvar, ymin, ymax))
```

Similarly, if given one function $f(x, y)$, then this function plots streamlines in the slope field $dy/dx = f(x, y)$ over the specified ranges as given above.

**PLOT OPTIONS:**

- `plot_points` – (default: 200) the minimal number of plot points
- `density` – float (default: 1.); controls the closeness of streamlines
- `start_points` – (optional) list of coordinates of starting points for the streamlines; coordinate pairs can be tuples or lists

**EXAMPLES:**

Plot some vector fields involving $\sin$ and $\cos$:

```python
sage: x, y = var('x y')
sage: streamline_plot((sin(x), cos(y)), (x,-3,3), (y,-3,3))
Graphics object consisting of 1 graphics primitive
```

```python
sage: streamline_plot((y, (cos(x)-2) * sin(x)), (x,-pi,pi), (y,-pi,pi))
Graphics object consisting of 1 graphics primitive
```

We increase the density of the plot:

```python
sage: streamline_plot((y, (cos(x)-2) * sin(x)), (x,-pi,pi), (y,-pi,pi), density=2)
Graphics object consisting of 1 graphics primitive
```
Chapter 2. Function and Data Plots
2.5. Streamline Plots
We ignore function values that are infinite or NaN:

```python
sage: x, y = var('x y')
sage: streamline_plot((-x/sqrt(x^2+y^2), -y/sqrt(x^2+y^2)), (x,-10,10), (y,-10,10))
Graphics object consisting of 1 graphics primitive
```

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

```python
sage: streamline_plot((x, y), (x,-2,2), (y,-2,2), xmax=10)
Graphics object consisting of 1 graphics primitive
sage: streamline_plot((x, y), (x,-2,2), (y,-2,2)).show(xmax=10) # These are equivalent
```

We can also construct streamlines in a slope field:

```python
sage: x, y = var('x y')
sage: streamline_plot((x + y) / sqrt(x^2 + y^2), (x,-3,3), (y,-3,3))
Graphics object consisting of 1 graphics primitive
```

We choose some particular points the streamlines pass through:

```python
sage: pts = [[1, 1], [-2, 2], [1, -3/2]]
sage: g = streamline_plot((x + y) / sqrt(x^2 + y^2), (x,-3,3), (y,-3,3), start_points=pts)
```

(continues on next page)
2.5. Streamline Plots
sage: g += point(pts, color='red')
sage: g
Graphics object consisting of 2 graphics primitives

Note: Streamlines currently pass close to start_points but do not necessarily pass directly through them. That is part of the behavior of matplotlib, not an error on your part.

2.6 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, alpha=1, markersize=50, marker='o', zorder=5, facecolor='#fec7b8', edgecolor='black', clip=True, aspect_ratio='automatic', **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"). See the documentation of plot() for the full list of markers.
- facecolor – default: '#fec7b8'
- edgecolor – default: 'black'
- zorder – default: 5

EXAMPLES:

sage: scatter_plot([[0,1],[2,4],[3.2,6]], marker='s')

Graphics object consisting of 1 graphics primitive

Extra options will get passed on to show(), as long as they are valid:
2.7 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 \((a, b)\)
- \( \text{vertical_lines} \) – 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)])
```

Graphics object consisting of 1 graphics primitive

```
sage: plot_step_function(([i, sin(i)) for i in range(5,20)])
```

Graphics object consisting of 1 graphics primitive

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)
```

Graphics object consisting of 14 graphics primitives

2.8 Histograms

```
class sage.plot.histogram.Histogram(datalist, options)
Bases: sage.plot.primitive.GraphicPrimitive
```

Graphics primitive that represents a histogram. This takes quite a few options as well.

**EXAMPLES:**

```
sage: from sage.plot.histogram import Histogram
sage: g = Histogram([1,3,2,0], {}); g
Histogram defined by a data list of size 4
sage: type(g)
<class 'sage.plot.histogram.Histogram'>
sage: opts = { 'bins':20, 'label':'mydata'}
sage: g = Histogram([random() for _ in range(500)], opts); g
Histogram defined by a data list of size 500
```

We can accept multiple sets of the same length:

```
sage: g = Histogram([[1,3,2,0], [4,4,3,3]], {}); g
Histogram defined by 2 data lists
```

**get_minmax_data ()**

Get minimum and maximum horizontal and vertical ranges for the Histogram object.

**EXAMPLES:**
sage: H = histogram([10,3,5], density=True); h = H[0]
sage: h.get_minmax_data() # rel tol 1e-15
{'xmax': 10.0, 'xmin': 3.0, 'ymax': 0.4761904761904765, 'ymin': 0}
sage: G = histogram([random() for _ in range(500)]); g = G[0]
sage: g.get_minmax_data() # random output
{'xmax': 0.99729312925213209, 'xmin': 0.00013024562219410285, 'ymax': 61, 'ymin': 0}
sage: Y = histogram([random()*10 for _ in range(500)], range=[2,8]); y = Y[0]
sage: ymm = y.get_minmax_data(); ymm['xmax'], ymm['xmin']
(8.0, 2.0)
sage: Z = histogram([[1,3,2,0], [4,4,3,3]]); z = Z[0]
sage: z.get_minmax_data()
{'xmax': 4.0, 'xmin': 0, 'ymax': 2, 'ymin': 0}

sage.plot.histogram.histogram(datalist, aspect_ratio='automatic', align='mid', weights=None, range=None, bins=10, edgecolor='black', **options)

Computes and draws the histogram for list(s) of numerical data. See examples for the many options; even more customization is available using matplotlib directly.

INPUT:

- **datalist** – A list, or a list of lists, of numerical data
- **align** – (default: “mid”) How the bars align inside of each bin. Acceptable values are “left”, “right” or “mid”
- **alpha** – (float in [0,1], default: 1) The transparency of the plot
- **bins** – The number of sections in which to divide the range. Also can be a sequence of points within the range that create the partition
- **color** – The color of the face of the bars or list of colors if multiple data sets are given
- **cumulative** – (boolean - default: False) If True, then a histogram is computed in which each bin gives the counts in that bin plus all bins for smaller values. Negative values give a reversed direction of accumulation
- **edgecolor** – The color of the border of each bar
- **fill** – (boolean - default: True) Whether to fill the bars
- **hatch** – (default: None) symbol to fill the bars with - one of `/`, `\`, `|`, `-`, `+`, `x`, `o`, `O`, `.`, `*`, `.` (or None)
- **hue** – The color of the bars given as a hue. See hue for more information on the hue
- **label** – A string label for each data list given
- **linewidth** – (float) width of the lines defining the bars
- **linestyle** – (default: ‘solid’) Style of the line. One of ‘solid’ or ‘-‘, ‘dashed’ or ‘–’, ‘dotted’ or ‘:’, ‘dashdot’ or ‘.’
- **density** – (boolean - default: False) If True, the result is the value of the probability density function at the bin, normalized such that the integral over the range is 1.
- **range** – A list [min, max] which define the range of the histogram. Values outside of this range are treated as outliers and omitted from counts
- **rwidth** – (float in [0,1], default: 1) The relative width of the bars as a fraction of the bin width
- **stacked** – (boolean - default: False) If True, multiple data are stacked on top of each other
• weights – (list) A sequence of weights the same length as the data list. If supplied, then each value contributes its associated weight to the bin count
• zorder – (integer) the layer level at which to draw the histogram

Note: The weights option works only with a single list. List of lists representing multiple data are not supported.

EXAMPLES:

A very basic histogram for four data points:

```
sage: histogram([1,2,3,4], bins=2)
Graphics object consisting of 1 graphics primitive
```

We can see how the histogram compares to various distributions. Note the use of the density keyword to guarantee the plot looks like the probability density function:

```
sage: nv = normalvariate
sage: H = histogram([nv(0,1) for _ in range(1000)], bins=20, density=True,
    range=[-5,5])
sage: P = plot( 1/sqrt(2*pi)*e^(-x^2/2), (x,-5,5), color='red', linestyle='--')
sage: H+P
Graphics object consisting of 2 graphics primitives
```

There are many options one can use with histograms. Some of these control the presentation of the data, even if it is boring:

```
sage: histogram(list(range(100)), color=(1,0,0), label='mydata',
    rwidth=.5, align="right")
Graphics object consisting of 1 graphics primitive
```

This includes many usual matplotlib styling options:

```
sage: T = RealDistribution('lognormal', [0,1])
sage: histogram( [T.get_random_element() for _ in range(100)], alpha=0.3,
    edgecolor='red', fill=False, linestyle='dashed', hatch='O', linewidth=5)
Graphics object consisting of 1 graphics primitive
```

We can do several data sets at once if desired:

```
sage: histogram(srange(0,1,.1)*10, [nv(0, 1) for _ in range(100)], color=['red',
    'green'], bins=5)
Graphics object consisting of 1 graphics primitive
```

We have the option of stacking the data sets too:

```
sage: histogram([ [1,1,1,1,2,2,2,3,3,3], [4,4,4,4,3,3,3,2,2,2] ], stacked=True,
    color=['blue', 'red'])
Graphics object consisting of 1 graphics primitive
```

It is possible to use weights with the histogram as well:

```
sage: histogram(list(range(10)), bins=3, weights=[1,2,3,4,5,5,4,3,2,1])
Graphics object consisting of 1 graphics primitive
```
2.9 Bar Charts

\textbf{class} \texttt{sage.plot.bar_chart.BarChart}(ind, datalist, options)  
\textbf{Bases:} \texttt{sage.plot.primitive.GraphicPrimitive}  

Graphics primitive that represents a bar chart.

\textbf{EXAMPLES:}

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

\texttt{get_minmax_data()}  
Returns a dictionary with the bounding box data.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: b = bar_chart([-2.3,5,-6,12])
d = b.get_minmax_data()
sage: d['xmin']
0
sage: d['xmax']
4
\end{verbatim}

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

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

\textbf{EXAMPLES:}

A bar\_chart with blue bars:

\begin{verbatim}
sage: bar_chart([1,2,3,4])
Graphics object consisting of 1 graphics primitive
\end{verbatim}

A bar\_chart with thinner bars:

\begin{verbatim}
sage: bar_chart([x^2 for x in range(1,20)], width=0.2)
Graphics object consisting of 1 graphics primitive
\end{verbatim}

A bar\_chart with negative values and red bars:

\begin{verbatim}
sage: bar_chart([-3,5,-6,11], rgbcolor=(1,0,0))
Graphics object consisting of 1 graphics primitive
\end{verbatim}

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

\begin{verbatim}
sage: bar_chart([-1,1,-1,1], legend_label='wave')
Graphics object consisting of 1 graphics primitive
\end{verbatim}

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

\begin{verbatim}
sage: bar_chart([-2,8,-7,3], rgbcolor=(1,0,0), axes=False)
Graphics object consisting of 1 graphics primitive
sage: bar_chart([-2,8,-7,3], rgbcolor=(1,0,0)).show(axes=False) # These are equivalent
\end{verbatim}
3.1 Graph Plotting

(For LaTeX drawings of graphs, see the graph_latex module.)

All graphs have an associated Sage graphics object, which you can display:

```python
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 \texttt{graphs.}\[\text{tab}\], the positions are preset. For example, consider the Petersen graph with default node positioning vs. the Petersen graph constructed by this database:

```
\begin{verbatim}
sage: petersen_spring = Graph('I`ES@obGkqegW~')
sage: petersen_spring.show() # long time
\end{verbatim}
```

```
\begin{verbatim}
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
\end{verbatim}
```

For all the constructors in this database (except some random graphs), the position dictionary is filled in, instead of using the spring-layout algorithm.

**Plot options**

Here is the list of options accepted by \texttt{plot()} and the constructor of \texttt{GraphPlot}. Those two functions also accept all options of \texttt{sage.plot.graphics.Graphics.show()}. 
| **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). |
| **iterations** | The number of times to execute the spring layout algorithm. |
| **heights** | A dictionary mapping heights to the list of vertices at this height. |
| **spring** | Use spring layout to finalize the current layout. |
| **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'. |
| **forest_roots** | An iterable specifying which vertices to use as roots for the layout='forest' option. If no root is specified for a tree, then one is chosen close to the center of the tree. Ignored unless layout='forest'. |
| **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. |
| **dim** | The dimension of the layout – 2 or 3. |
| **prog** | Which graphviz layout program to use – one of “circo”, “dot”, “fdp”, “neato”, or “twopi”. |
| **by_component** | Whether to do the spring layout by connected component – a boolean. |
| **pos** | The position dictionary of vertices |
| **vertex_labels** | Whether or not to draw vertex labels. |
| **vertex_color** | Default color for vertices not listed in vertex_colors dictionary. |
| **vertex_colors** | Dictionary of vertex coloring : each key is a color recognizable by matplotlib, and each corresponding entry is a list of vertices. |
| **vertex_size** | The size to draw the vertices. |
| **vertex_shape** | The shape to draw the vertices. Currently unavailable for Multi-edged Di-Graphs. |
| **edge_labels** | Whether or not to draw edge labels. |
| **edge_style** | The linestyle of the edges. It should be one of “solid”, “dashed”, “dotted”, dashdot”, or “-”, “--”, “:”, “-.”, respectively. |
| **edge_thickness** | The thickness of the edges. |
| **edge_color** | The default color for edges not listed in edge_colors. |
| **edge_colors** | a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges. |
| **color_by_label** | Whether to color the edges according to their labels. This also accepts a function or dictionary mapping labels to colors. |
| **partition** | A partition of the vertex set. If specified, plot will show each cell in a different color. vertex_colors takes precedence. |
| **loop_size** | The radius of the smallest loop. |
| **dist** | The distance between multiedges. |
| **max_dist** | The max distance range to allow multiedges. |
| **talk** | Whether to display the vertices in talk mode (larger and white). |
| **graph_border** | Whether or not to draw a frame around the graph. |
| **edge_labels_background** | The color of the background of the edge labels |

**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):

```python
sage: import sage.graphs.graph_plot
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:

```python
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:

```python
sage: import sage.graphs.graph_plot
sage: sage.graphs.graph_plot.DEFAULT_SHOW_OPTIONS['figsize'] = [4,4]
```

Index of methods and functions

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>GraphPlot.set_pos()</code></td>
<td>Set the position plotting parameters for this GraphPlot.</td>
</tr>
<tr>
<td><code>GraphPlot.set_vertices()</code></td>
<td>Set the vertex plotting parameters for this GraphPlot.</td>
</tr>
<tr>
<td><code>GraphPlot.set_edges()</code></td>
<td>Set the edge (or arrow) plotting parameters for the GraphPlot object.</td>
</tr>
<tr>
<td><code>GraphPlot.show()</code></td>
<td>Show the (Di)Graph associated with this GraphPlot object.</td>
</tr>
<tr>
<td><code>GraphPlot.plot()</code></td>
<td>Return a graphics object representing the (di)graph.</td>
</tr>
<tr>
<td><code>GraphPlot.layout_tree()</code></td>
<td>Compute a nice layout of a tree.</td>
</tr>
</tbody>
</table>

```python
class sage.graphs.graph_plot.GraphPlot(graph, options)
    Bases: sage.structure.sage_object.SageObject

    Return 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.

    EXAMPLES:
```


```python
....: '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,
....: 'edge_labels_background': 'transparent'}
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:

```python
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)
Return 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:

```python
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':[4,9]}
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]
....: for i in range(5,10):
```

(continues on next page)
Here are some more common graphs with typical options:

```python
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

```python
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:

```python
sage: D = DiGraph({ 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4],
....:     4: [17, 5], 5: [6, 15], 6: [7], 7: [8, 14], 8: [9], 9: [10, 13],
....:     10: [11], 11: [12, 18], 12: [16, 13], 13: [14], 14: [15], 15: [16],
```

(continues on next page)
3.1. Graph Plotting
This example shows off the coloring of edges:

```python
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:

```python
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]]
```

(continues on next page)
Loops are also plotted correctly:

```python
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:

```python
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()
```

Graphics object consisting of 11 graphics primitives
3.1. Graph Plotting
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()
Graphics object consisting of 14 graphics primitives
```

```
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()
Graphics object consisting of 14 graphics primitives
```

```
sage: t.set_edge_label(0,1,-7)
sage: t.set_edge_label(0,5,3)
sage: t.set_edge_label(0,5,99)
```
(continues on next page)
3.1. Graph Plotting
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()
Graphics object consisting of 20 graphics primitives

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()
Graphics object consisting of 34 graphics primitives

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,'g'),(2,1,'h')])
sage: g.graphplot(edge_labels=True, color_by_label=True, edge_style='dashed').plot()
Graphics object consisting of 26 graphics primitives

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()
Graphics object consisting of 26 graphics primitives

**set_edges(**edge_options)

Set 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.
3.1. Graph Plotting
EXAMPLES:

```python
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='dashed')
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```

```python
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='dashed')
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
Graphics object consisting of 28 graphics primitives
```

```python
sage: GP.set_edges(edge_color='black')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
```
3.1. Graph Plotting
Chapter 3. Plots of Other Mathematical Objects
**set_pos()**

Set the position plotting parameters for this GraphPlot.

**EXAMPLES:**

This function is called implicitly by the code below:

```python
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:

```python
sage: g.get_pos()
{0: (0.0, 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]})
Graphics object consisting of 14 graphics primitives

\texttt{set\_vertices(**vertex\_options)}

Set the vertex plotting parameters for this \texttt{GraphPlot}.

This function is called by the constructor but can also be called to make updates to the vertex options of an existing \texttt{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='dashed')
sage: GP.set_vertices(talk=True)
sage: GP.plot()
Graphics object consisting of 26 graphics primitives
sage: GP.set_vertices(vertex_color='green', vertex_shape='^')
sage: GP.plot()
Graphics object consisting of 26 graphics primitives

\texttt{show(**kwds)}
Show 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.

EXAMPLES:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.graphplot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```
3.2 Matrix Plots

```python
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). If None, the defaults are used as indicated in matrix_plot().
    - `yrange` - tuple of 2 floats indicating range for vertical direction (number of rows in the matrix). If None, the defaults are used as indicated in matrix_plot().
    - `options` - dict of valid plot options to pass to constructor

    EXAMPLES:

    Note this should normally be used indirectly via matrix_plot():

    ```python
    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:

    ```python
    sage: matrix_plot([[1, 0], [0, 1]], fontsize=10)
    Graphics object consisting of 1 graphics primitive
    sage: matrix_plot([[1, 0], [0, 1]]).show(fontsize=10)  # These are equivalent
    ```

    `get_minmax_data()`

    Returns a dictionary with the bounding box data.

    EXAMPLES:

    ```python
    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', 2.5), ('ymin', -0.5)]
    ```
```

sage.plot.matrix_plot.matrix_plot(mat, xrange=None, yrange=None, aspect_ratio=1, axes=False, cmap='Greys', colorbar=False, frame=True, marker='.', norm=None, flip_y=True, subdivisions=False, ticks_integer=True, vmin=None, vmax=None, subdivision_boundaries=False, subdivision_style=None, colorbar_orientation='vertical', colorbar_format=None, **options)

A plot of a given matrix or 2D array.

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.
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.

The default is for the lowest number to be black and the highest number to be white in a greyscale pattern; see the information about normalizing below. To reverse this, use `cmap='Greys'`.

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
- `xrange` - (default: None) tuple of the horizontal extent \((x_{\text{min}}, x_{\text{max}})\) of the bounding box in which to draw the matrix. The image is stretched individually along \(x\) and \(y\) to fill the box.
  
  If None, the extent is determined by the following conditions. Matrix entries have unit size in data coordinates. Their centers are on integer coordinates, and their center coordinates range from 0 to columns-1 horizontally and from 0 to rows-1 vertically.
  
  If the matrix is sparse, this keyword is ignored.
- `yrange` - (default: None) tuple of the vertical extent \((y_{\text{min}}, y_{\text{max}})\) of the bounding box in which to draw the matrix. See `xrange` for details.

The following input must all be passed in as named parameters, if default not used:

- `cmap` - a colormap (default: ‘Greys’), the name of a predefined colormap, a list of colors, or an instance of a matplotlib Colormap.
  
  The list of predefined color maps can be visualized in matplotlib's documentation. You can also type `import matplotlib.cm; matplotlib.cm.datad.keys()` to list their 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)
- `flip_y` - (default: True) boolean. If False, the first row of the matrix is on the bottom of the graph. Otherwise, the first row is on the top of the graph.
- `subdivisions` - If True, plot the subdivisions of the matrix as lines.
- `subdivision_boundaries` - a list of lists in the form \([\text{row}_{\text{subdivisions}}, \text{column}_{\text{subdivisions}}]\), 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 $\mathbb{Z}$ colored with different grey levels:

```sage
sage: matrix_plot(matrix([[1,3,5,1],[2,4,5,6],[1,3,5,7]]))
```

Graphics object consisting of 1 graphics primitive

Here we make a random matrix over $\mathbb{R}$ and use cmap='hsv' to color the matrix elements different RGB colors:

```sage
sage: matrix_plot(random_matrix(RDF, 50), cmap='hsv')
```

Graphics object consisting of 1 graphics primitive

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
sage: P = matrix_plot(matrix(2,[1,1,3,3]))
sage: Q = matrix_plot(matrix(2,[2,2,3,3]))
sage: P; Q
```

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

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:
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=[dict(color='red', thickness=3), dict(linestyle='--',
               thickness=6)])
```

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 `flip_y` argument:

```
sage: matrix_plot(identity_matrix(100), flip_y=False)
```

A custom bounding box in which to draw the matrix can be specified using the `xrange` and `yrange` arguments:

```
sage: P = matrix_plot(identity_matrix(10), xrange=(0, pi), yrange=(-pi, 0))
sage: P
```

If the horizontal and vertical dimension of the image are very different, the default `aspect_ratio=1` may be unsuitable and can be changed to `automatic`:

```
sage: matrix_plot(random_matrix(RDF, 2, 2), (-100, 100), (0, 1),
               aspect_ratio='automatic')
```

Another random plot, but over $F_{389}$:

```
sage: m = random_matrix(GF(389), 10)
sage: matrix_plot(m, cmap='Oranges')
```

It also works if you lift it to the polynomial ring:
We have several options for colorbars:

```
sage: matrix_plot(random_matrix(RDF, 50), colorbar=True, colorbar_orientation='horizontal')
Graphics object consisting of 1 graphics primitive
```

```
sage: matrix_plot(random_matrix(RDF, 50), colorbar=True, colorbar_format='%.3f')
Graphics object consisting of 1 graphics primitive
```

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})
Graphics object consisting of 1 graphics primitive
```

Here we plot a random sparse matrix:

```
sage: sparse = matrix(dict([(randint(0, 10), randint(0, 10)), 1) for i in range(100)]))
sage: matrix_plot(sparse)
Graphics object consisting of 1 graphics primitive
```

```
sage: A=random_matrix(ZZ,100000,density=.00001,sparse=True)
sage: matrix_plot(A,marker=',')
Graphics object consisting of 1 graphics primitive
```

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)
Graphics object consisting of 1 graphics primitive
```

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))
Graphics object consisting of 1 graphics primitive
```

Plotting lists of lists also works:
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), flip_y=False, title='not identity')
```

The title position is adjusted upwards if the `flip_y` keyword is set to True (this is the default):

```
sage: matrix_plot(identity_matrix(50), title='identity')
```
4.1 Arcs of circles and ellipses

```python
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 0.78539816339... inside the sector (0.78539816339...,1.5707963267...)
```

```python
bezier_path() -> Return self as a Bezier path.

This is needed to concatenate arcs, in order to create hyperbolic polygons.

EXAMPLES:

```
**get_minmax_data()**

Return 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()**

```
sage.plot.arc.arc(center, r1, r2=None, angle=0.0, sector=0.0, 6.283185307179586, alpha=1, thickness=1, linestyle='solid', zorder=5, rgbcolor='blue', aspect_ratio=1.0, **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 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))
Graphics object consisting of 1 graphics primitive
```

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

```
sage: arc((2,3), 2, 1, sector=(0,pi/2))
Graphics object consisting of 1 graphics primitive
```

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))
Graphics object consisting of 1 graphics primitive
```

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")
Graphics object consisting of 1 graphics primitive
```

```
sage: arc((0,0), 2, 1, 0, (0,pi/2), linestyle="--", color="red")
Graphics object consisting of 1 graphics primitive
```

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
```

### 4.2 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:**

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

```python
plot3d(tail=0, zhead=0, **kwds)
```
Takes 2D plot and places it in 3D.

**EXAMPLES:**

```python
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 `Graphics.plot3d()` method, but only as a whole:

```python
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:

```python
sage: A = arrow((0,0),(1,1))[0].plot3d(3,4)
```

```python
class sage.plot.arrow.CurveArrow(path, options)
```
Bases: `sage.plot.primitive.GraphicPrimitive`

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

**EXAMPLES:**

```python
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)
```

```python
get_minmax_data()
```
Returns a dictionary with the bounding box data.

**EXAMPLES:**

```python
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
```

```python
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:

```python
sage: arrow((0,0), (1,1))
Graphics object consisting of 1 graphics primitive
```

```python
sage: arrow((0,0,1), (1,1,1))
Graphics3d Object
```

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

If `tailpoint` and `headpoint` are provided, returns an arrow from (x\text{tail}, y\text{tail}) to (x\text{head}, y\text{head}). 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:

```python
element = arrow2d((1,1), (3,3))
```

Graphics object consisting of 1 graphics primitive

Make a red arrow:
You can change the width of an arrow:

```sage
sage: arrow2d((1,1), (3,3), width=5, arrowsize=15)
sage: arrow2d((1,1), (3,3), linestyle='dashed')
sage: arrow2d((1,1), (3,3), linestyle='--')
```

A pretty circle of arrows:

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

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 1 graphics primitive

Graphics object consisting of 63 graphics primitives
4.2. Arrows
4.2. Arrows
If we want to draw the arrow between objects, for example, the boundaries of two lines, we can use the `arrowshorten` option to make the arrow shorter by a certain number of points:

```sage
sage: L1 = line([(0,0), (1,0)], thickness=10)
sage: L2 = line([(0,1), (1,1)], thickness=10)
sage: A = arrow2d((0.5,0), (0.5,1), arrowshorten=10, rgbcolor=(1,0,0))
sage: L1 + L2 + A
Graphics object consisting of 3 graphics primitives
```

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

```sage
sage: arrow2d(headpoint=None, tailpoint=None)
Graphics object consisting of 0 graphics primitives
```

We can also draw an arrow with a legend:

```sage
sage: arrow((0,0), (0,2), legend_label='up', legend_color='purple')
Graphics object consisting of 1 graphics primitive
```

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

```sage
sage: arrow2d((-2,2), (7,1), frame=True)
Graphics object consisting of 1 graphics primitive
```

4.2. Arrows
4.2. Arrows
4.3 Bezier Paths

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

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:

```python
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:

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

Graphics object consisting of 1 graphics primitive
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
Graphics3d Object
```
sage: bezier3d(((0,0,0),(1,0,0),(0,1,0),(0,1,1)))
Graphics3d Object

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:

\[
\text{path} = [[p_1, c_1, c_2, p_2], [c_3, c_4, p_3], [c_5, p_4], [p_5], \ldots]
\]

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 \(p_1\) and \(p_2\) is tangent to the line through \(p_1\) and \(c_1\) at \(p_1\), and tangent to the line through \(p_2\) and \(c_2\) at \(p_2\). Similarly, the curve between \(p_2\) and \(p_3\) is tangent to line(p2,c3) at \(p_2\) and tangent to line(p3,c4) at \(p_3\). Curve(p3,p4) is tangent to line(p3,c5) at \(p_3\) and tangent to line(p4,c5) at \(p_4\). Curve(p4,p5) is a straight line.

INPUT:
4.3. Bezier Paths
• **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.
• **rgbcolor** – default: (0,0,0)
• **zorder** – the layer in which to draw

**EXAMPLES:**

```python
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
Graphics object consisting of 1 graphics primitive
```

To construct a simple curve, create a list containing a single list:

```python
sage: path = [[(0,0),(.5,1),(1,0)]]
sage: curve = bezier_path(path, linestyle='dashed', rgbcolor='green')
sage: curve
Graphics object consisting of 1 graphics primitive
```
4.3. Bezier Paths
Extra options will get passed on to \texttt{show()}, as long as they are valid:

\begin{verbatim}
\texttt{sage: bezier_path([[0,1],(.5,0),(1,1)], fontsize=50)}
Graphics object consisting of 1 graphics primitive
\texttt{sage: bezier_path([[0,1],(.5,0),(1,1)]).show(fontsize=50)} # These are equivalent
\end{verbatim}

4.4 Circles

\begin{verbatim}
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 \texttt{circle}:
\end{verbatim}
```
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
```

**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()
Graphics3d Object
```

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)])
Graphics3d Object
```

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
0.3
```

```
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())[-1]
'color $line_1 translucent 0.7 [204,0,255]'
```

```
sage.plot.circle.circle(center, radius, alpha=1, fill=False, thickness=1, edgecolor='blue', facecolor='blue', linestyle='solid', zorder=5, legend_label=None, legend_color=None, clip=True, aspect_ratio=1.0, **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
Graphics object consisting of 1 graphics primitive
```
We can also use this command to plot three-dimensional circles parallel to the $xy$-plane:

\begin{verbatim}
sage: c = circle((1,1,3), 1, rgbcolor=(1,0,0))
sage: c
Graphics3d Object
type(c)
<class 'sage.plot.plot3d.base.TransformGroup'>
\end{verbatim}

To correct the aspect ratio of certain graphics, it is necessary to show with a figsize of square dimensions:

\begin{verbatim}
sage: c.show(figsize=[5,5], xmin=-1, xmax=3, ymin=-1, ymax=3)
\end{verbatim}

Here we make a more complicated plot, with many circles of different colors:

\begin{verbatim}
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/step, PI/step)]:
\end{verbatim}

(continues on next page)
4.4. Circles
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', facecolor='red')
Graphics object consisting of 1 graphics primitive
```

This produces an all-green filled circle:

```
sage: circle((2,3), 1, fill=True, edgecolor='blue', rgbcolor='green')
Graphics object consisting of 1 graphics primitive
```

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)
Graphics object consisting of 1 graphics primitive
```

And circles with legends:
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!
```
Graphics object consisting of 1 graphics primitive

```
sage: circle((0, 0), 2).show(figsize=[10,10])  # These are equivalent
```
4.4. Circles
4.5 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 (1.5707963267..., 3.1415926535...) radians
    sage: D.options()['zorder']
    3
    sage: D.x
    1.0

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:
These examples show that the appropriate options are passed:

```python
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
0.3
```

```python
define: D = disk((2,3), 1, (pi/4,pi/3), hue=.8, alpha=.3, fill=False)
define: d = D[0]
define: dd = d.plot3d(z=2)
define: dd.jmol_repr(dd.testing_render_params())[0][-1]
'color $line_4 translucent 0.7 [204,0,255]'
```

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:

```python
define: bl = disk((0.0,0.0), 1, (pi, 3*pi/2), color='yellow')
define: tr = disk((0.0,0.0), 1, (0, pi/2), color='yellow')
define: tl = disk((0.0,0.0), 1, (pi/2, pi), color='black')
define: br = disk((0.0,0.0), 1, (3*pi/2, 2*pi), color='black')
define: P = tl+tr+bl+br
define: P.show(xmin=-2,xmax=2,ymin=-2,ymax=2)
```

The default aspect ratio is 1.0:

```python
sage: disk((0.0,0.0), 1, (pi, 3*pi/2)).aspect_ratio()
1.0
```

Another example of a disk:

```python
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`:

```python
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:

```python
4.5. Disks
```
```python
sage: d = disk((1,1,3), 1, (pi,3*pi/2), rgbcolor=(1,0,0))
sage: d
Graphics3d Object
sage: type(d)
<... 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

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

```python
sage: disk((0, 0), 5, (0, pi/2), xmin=0, xmax=5, ymin=0, ymax=5, figsize=(2,2),
       →rgbcolor=(1, 0, 1))
Graphics object consisting of 1 graphics primitive
sage: disk((0, 0), 5, (0, pi/2), rgbcolor=(1, 0, 1)).show(xmin=0, xmax=5, ymin=0,
       →ymax=5, figsize=(2,2)) # These are equivalent
```

## 4.6 Ellipses

```python
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`:

```python
sage: from sage.plot.ellipse import Ellipse
sage: Ellipse(0, 0, 2, 1, pi/4, {})
```

**get_minmax_data()**

Return a dictionary with the bounding box data.

The bounding box is computed to be as minimal as possible.

**EXAMPLES:**

An example without an angle:

```python
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 $\pi/2$:

```python
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
```


4.6. Ellipses
We see that \texttt{rgbcolor} overrides these other options, as this plot is green:

\begin{verbatim}
sage: ellipse((0,0),3,1,pi/6,fill=True,edgecolor='black',facecolor='red',rgbcolor='green')
Graphics object consisting of 1 graphics primitive
\end{verbatim}

The default aspect ratio for ellipses is 1.0:

\begin{verbatim}
sage: ellipse((0,0),2,1).aspect_ratio()
1.0
\end{verbatim}

One cannot yet plot ellipses in 3D:

\begin{verbatim}
sage: ellipse((0,0,0),2,1)
Traceback (most recent call last):
... 
NotImplementedError: plotting ellipse in 3D is not implemented
\end{verbatim}

We can also give ellipses a legend:

\begin{verbatim}
sage: ellipse((0,0),2,1,legend_label="My ellipse", legend_color='green')
Graphics object consisting of 1 graphics primitive
\end{verbatim}

## 4.7 Line Plots

### class \texttt{sage.plot.line.Line}(xdata, ydata, options)

Bases: \texttt{sage.plot.primitive.GraphicPrimitive_xydata}

Primitive class that initializes the line graphics type.

**EXAMPLES:**

\begin{verbatim}
sage: from sage.plot.line import Line
sage: Line([(1,2,7), (1,5,-1), {}])
Line defined by 3 points
\end{verbatim}

\texttt{plot3d}(z=0, **\texttt{kwds})

Plots a 2D line in 3D, with default height zero.

**EXAMPLES:**

\begin{verbatim}
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)
Graphics3d Object
\end{verbatim}

\texttt{sage.plot.line.line}(points, **\texttt{kwds})

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

**INPUT:**

- \texttt{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:

```python
sage: line([(0,0), (1,1)])
Graphics object consisting of 1 graphics primitive
```

```python
graphics. line([(0,0,1), (1,1,1)])
Graphics3d Object
```

```
sage.plot.line.line2d(points, alpha=1, rgbcolor=0, 0, 1, thickness=1, legend_label=None, legend_color=None, aspect_ratio='automatic', **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: "-" or "solid") 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
Graphics object consisting of 0 graphics primitives
sage: import numpy; line(numpy.array([]))
Graphics object consisting of 0 graphics primitives
sage: line([(1,1)])
Graphics object consisting of 1 graphics primitive
```

A line with numpy arrays:

```
sage: line(numpy.array(([1,2],[3,4])))
Graphics object consisting of 1 graphics primitive
```

A line with a legend:

```
sage: line([(0,0),(1,1)], legend_label='line')
Graphics object consisting of 1 graphics primitive
```

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
Graphics object consisting of 2 graphics primitives
```

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

```
sage: line([(0,1), (3,4)], figsize=[10, 2])
Graphics object consisting of 1 graphics primitive
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)
Graphics object consisting of 1 graphics primitive
```
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(1,100)]
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 in range(200)]
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 in range(200)]
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: 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 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)^2), 10*tan(-pi/2+pi*i/100)*(1-4*cos(-pi/2+pi*i/100)^2)) for i in range(1,100)]
sage: line(L, rgbcolor=(1/4,1/4,1/2))
```

A red plot of the Jacobi elliptic function 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))
```

4.7. Line Plots 363
A red plot of $J$-Bessel function $J_2(x)$, $0 < x < 10$:

```python
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))
```
Graphics object consisting of 1 graphics primitive

A purple plot of the Riemann zeta function $\zeta(1/2 + it)$, $0 < t < 30$:

```python
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))
```
Graphics object consisting of 1 graphics primitive

A purple plot of the Hasse-Weil $L$-function $L(E, 1 + it)$, $-1 < t < 10$:

```python
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))
```
Graphics object consisting of 1 graphics primitive

A red, blue, and green “cool cat”:

```python
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
```
Graphics object consisting of 3 graphics primitives

### 4.8 Points

```python
class sage.plot.point.Point(xdata, ydata, options)
```

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:

```python
sage: from sage.plot.point import Point
```
```
sage: P = Point([1,2],[2,3],'alpha':.5)
```
```
sage: P.options()['alpha']
```
0.500000000000000
```
```
sage: P.xdata
```
[1, 2]
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:

```python
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:

```python
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:

```python
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:

```python
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:

```python
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
```

`sage.plot.point.point(points, **kwds)`
Returns either a 2-dimensional or 3-dimensional point or sum of points.

4.8. 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()`.

See also:

`sage.plot.point.point2d()`, `sage.plot.plot3d.shapes2.point3d()`

EXAMPLES:

```python
sage: point((1,2))
Graphics object consisting of 1 graphics primitive
```

```python
sage: point((1,2,3))
Graphics3d Object
```

```python
sage: point([(0,0), (1,1)])
Graphics object consisting of 1 graphics primitive
```

```python
sage: point([(0,0,1), (1,1,1)])
Graphics3d Object
```

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

```python
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)], frame=True)
Graphics object consisting of 1 graphics primitive
```

```python
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)]).show(frame=True)  # These are equivalent
```

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.
- **alpha** – How transparent the point is.
- **faceted** – If True color the edge of the point. (only for 2D plots)
- **hue** – The color given as a hue.
- **legend_color** – The color of the legend text
- **legend_label** – The label for this item in the legend.
- **marker** – the marker symbol for 2D plots only (see documentation of `plot()` for details)
- **markeredgecolor** – the color of the marker edge (only for 2D plots)
- **rgbcolor** – The color as an RGB tuple.
- **size** – How big the point is (i.e., area in points^2=(1/72 inch)^2).
- **zorder** – The layer level in which to draw
EXAMPLES:

A purple point from a single tuple or coordinates:

```
sage: point((0.5, 0.5), rgbcolor=hue(0.75))
Graphics object consisting of 1 graphics primitive
```

Points with customized markers and edge colors:

```
sage: r = [(random(), random()) for _ in range(10)]
sage: point(r, marker='d', markeredgecolor='red', size=20)
Graphics object consisting of 1 graphics primitive
```

Passing an empty list returns an empty plot:

```
sage: point([])
Graphics object consisting of 0 graphics primitives
sage: import numpy; point(numpy.array([]))
Graphics object consisting of 0 graphics primitives
```

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)
Graphics object consisting of 1 graphics primitive
```

And an example with a legend:

```
sage: point((0,0), rgbcolor='black', pointsize=40, legend_label='origin')
Graphics object consisting of 1 graphics primitive
```

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')
Graphics object consisting of 2 graphics primitives
```

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)
Graphics object consisting of 1 graphics primitive
sage: point([(cos(theta), sin(theta)) for theta in srange(0, 2*pi, pi/8)]).show(frame=True) # These are equivalent
```
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)
Graphics object consisting of 1 graphics primitive
```

Since Sage Version 4.4 (trac 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)
Graphics object consisting of 1 graphics primitive
```

```
sage: point((3,4), pointsize=100)
Graphics object consisting of 1 graphics primitive
```

We can plot a single complex number:

```
sage: point(CC(1+I), pointsize=100)
Graphics object consisting of 1 graphics primitive
```

We can also plot a list of complex numbers:

```
sage: point([CC(I), CC(I+1), CC(2+2*I)], pointsize=100)
Graphics object consisting of 1 graphics primitive
```

`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`.

**See also:**

`sage.plot.point.point2d()`, `sage.plot.plot3d.shapes2.point3d()`

**EXAMPLES:**

```
sage: point((1,2))
Graphics object consisting of 1 graphics primitive
```

```
sage: point((1,2,3))
Graphics3d Object
```

```
sage: point([(0,0), (1,1)])
Graphics object consisting of 1 graphics primitive
```

```
sage: point([(0,0,1), (1,1,1)])
Graphics3d Object
```

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

class sage.plot.polygon.Polygon(xdata, ydata, options)

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]

plot3d(z=0, **kws)

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()
Graphics3d Object

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']

(continues on next page)
sage.plot.polygon.\texttt{polygon}(\texttt{points}, **\texttt{options})

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

For information regarding additional arguments, see either \texttt{polygon2d()} or \texttt{polygon3d()}. Options may be found and set using the dictionaries \texttt{polygon2d.options} and \texttt{polygon3d.options}.

EXAMPLES:

\begin{verbatim}
sage: polygon([(0,0), (1,1), (0,1)])
Graphics object consisting of 1 graphics primitive
\end{verbatim}
sage: polygon([(0,0), (1,1), (0,1)], axes=False)
Graphics object consisting of 1 graphics primitive
sage: polygon([(0,0), (1,1), (0,1)]).show(axes=False) # These are equivalent

sage.plot.polygon.polygon2d(points, alpha=1, rgbcolor=0, 0, 1, edgecolor=None, thickness=None, legend_label=None, legend_color=None, aspect_ratio=1.0, 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))
Graphics object consisting of 1 graphics primitive

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)
Graphics object consisting of 1 graphics primitive

In either case, the thickness of the border can be controlled:
For filled polygons, one can use different colors for the border and the interior as follows:

```
sage: L = [[0,0]]+[[i/100, 1.1+cos(i/20)] for i in range(100)]+[[1,0]]
sage: polygon2d(L, color="limegreen", edgecolor="black", axes=False)
```

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:
4.9. Polygons
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 in range(200)]
sage: polygon2d(L, rgbcolor=(1/8,1/4,1/2))
```
Graphics object consisting of 1 graphics primitive

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 in range(200)]
sage: polygon2d(L, rgbcolor=(1/8,1/4,3/4))
```
Graphics object consisting of 1 graphics primitive

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 in range(200)]
```
(continues on next page)
Chapter 4. Basic Shapes
A brown astroid:

\begin{verbatim}
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))
\end{verbatim}

And, my favorite, a greenish blob:

\begin{verbatim}
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))
\end{verbatim}

This one is for my wife:

\begin{verbatim}
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))
\end{verbatim}

One can do the same one with a colored legend label:
4.9. Polygons
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.

## 4.10 Arcs in hyperbolic geometry

AUTHORS:

- Hartmut Monien (2011 - 08)

```python
class sage.plot.hyperbolic_arc.HyperbolicArc(A, B, options)
    Bases: sage.plot.bezier_path.BezierPath

Primitive class for hyperbolic 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:

```python
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, alpha=1, fill=False, thickness=1, rgbcolor='blue', zorder=2, linestyle='solid', **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 by 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:

```python
sage: hyperbolic_arc(0, 1)
Graphics object consisting of 1 graphics primitive
```

Show a hyperbolic arc from 1/2 to i with a red thick line:

```python
sage: hyperbolic_arc(1/2, I, color='red', thickness=2)
Graphics object consisting of 1 graphics primitive
```

Show a hyperbolic arc from i to 2i with dashed line:

```python
sage: hyperbolic_arc(I, 2*I, linestyle='dashed')
Graphics object consisting of 1 graphics primitive
sage: hyperbolic_arc(I, 2*I, linestyle='--')
Graphics object consisting of 1 graphics primitive
```
4.11 Polygons and triangles in hyperbolic geometry

AUTHORS:
- Hartmut Monien (2011-08)
- Vincent Delecroix (2014-11)

```python
class sage.plot.hyperbolic_polygon.HyperbolicPolygon(pts, options)

Bases: sage.plot.bezier_path.BezierPath

Primitive class for hyperbolic polygon type.

See `hyperbolic_polygon?` for information about plotting a hyperbolic polygon in the complex plane.

INPUT:
- `pts` – coordinates of the polygon (as complex numbers)
- `options` – dict of valid plot options to pass to constructor

EXAMPLES:
Note that constructions should use `hyperbolic_polygon()` or `hyperbolic_triangle()`:

```python
sage: from sage.plot.hyperbolic_polygon import HyperbolicPolygon
sage: print(HyperbolicPolygon([0, 1/2, I], {}))
Hyperbolic polygon (0.000000000000000, 0.500000000000000, 1.000000000000000*I)
```

sage.plot.hyperbolic_polygon.hyperbolic_polygon(pts, alpha=1, fill=False, thickness=1, rgbcolor='blue', zorder=2, linestyle='solid', **options)

Return a hyperbolic polygon in the hyperbolic plane with vertices `pts`.

Type `?hyperbolic_polygon` to see all options.

INPUT:
- `pts` – a list or tuple of complex numbers

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 polygon with coordinates $-1, 3i, 2+2i, 1+i$:

```python
sage: hyperbolic_polygon([-1,3*I,2+2*I,1+I])
Graphics object consisting of 1 graphics primitive
```

With more options:

```python
sage: hyperbolic_polygon([-1,3*I,2+2*I,1+I], fill=True, color='red')
Graphics object consisting of 1 graphics primitive
```
4.11. Polygons and triangles in hyperbolic geometry
sage.plot.hyperbolic_polygon.hyperbolic_triangle(a, b, c, **options)

Return a hyperbolic triangle in the hyperbolic plane with vertices (a, b, c).

Type ?hyperbolic_polygon 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 + \( \frac{i\sqrt{3}}{2} \) and \(-1/2 + \frac{i\sqrt{3}}{2}\):

```
sage: hyperbolic_triangle(0, -1/2+I*sqrt(3)/2, 1/2+I*sqrt(3)/2)
Graphics object consisting of 1 graphics primitive
```

A hyperbolic triangle with coordinates 0, 1 and \(2 + i\) and a dashed line:
4.12 Regular polygons in the upper half model for hyperbolic plane

AUTHORS:

- Javier Honrubia (2016-01)

**class** `sage.plot.hyperbolic_regular_polygon.HyperbolicRegularPolygon` *(sides, i_angle, center, options)*

Bases: `sage.plot.hyperbolic_polygon.HyperbolicPolygon`

Primitive class for regular hyperbolic polygon type.

See `hyperbolic_regular_polygon?` for information about plotting a hyperbolic regular polygon in the upper complex halfplane.

**INPUT:**

- `sides` – number of sides of the polygon
- `i_angle` – interior angle of the polygon
• center—center point as a complex number of the polygon

EXAMPLES:

Note that constructions should use `hyperbolic_regular_polygon()`:

```
sage: from sage.plot.hyperbolic_regular_polygon import HyperbolicRegularPolygon
sage: print(HyperbolicRegularPolygon(5, pi/2, I, {}))
Hyperbolic regular polygon (sides=5, i_angle=1/2*pi, center=1.0000000000000000000*I)
```

The code verifies there exists a compact hyperbolic regular polygon with the given data, checking

\[ A(P) = \pi(s - 2) - s \cdot \alpha > 0, \]

where \( s \) is sides and \( \alpha \) is \( \text{i_angle} \). This raises an error if the \( \text{i_angle} \) is less than the minimum to generate a compact polygon:

```
sage: from sage.plot.hyperbolic_regular_polygon import HyperbolicRegularPolygon
sage: P = HyperbolicRegularPolygon(4, pi/2, I, {})
Traceback (most recent call last):
  ... ValueError: there exists no hyperbolic regular compact polygon,
for sides=4 the interior angle must be less than 1/2*pi
```

It is an error to give a center outside the upper half plane in this model

```
sage: from sage.plot.hyperbolic_regular_polygon import HyperbolicRegularPolygon
sage: P = HyperbolicRegularPolygon(4, pi/4, 1-I, {})
Traceback (most recent call last):
  ... ValueError: center: 1.00000000000000 - 1.00000000000000*I is not
a valid point in the upper half plane model of the hyperbolic plane
```

Return a hyperbolic regular polygon in the upper half model of Hyperbolic plane given the number of sides, interior angle and possibly a center.

Type `?hyperbolic_regular_polygon` to see all options.

INPUT:

• sides—number of sides of the polygon
• i_angle—interior angle of the polygon
• center—(default: `i`) hyperbolic center point (complex number) of the polygon

OPTIONS:

• alpha—default: 1
• fill—default: True
- **thickness** – default: 1
- **rgbcolor** – default: 'blue'
- **linestyle** – (default: 'solid') the style of the line, which can be one of the following:
  - 'dashed' or '--'
  - 'dotted' or ':'
  - 'solid' or '-'
  - 'dashdot' or '-.'

**EXAMPLES:**

Show a hyperbolic regular polygon with 6 sides and square angles:

```python
sage: g = hyperbolic_regular_polygon(6, pi/2)
sage: g.plot()
```

Graphics object consisting of 1 graphics primitive

With more options:

```python
sage: g = hyperbolic_regular_polygon(6, pi/2, center=3+2*I, fill=True, color='red')
sage: g.plot()
```

Graphics object consisting of 1 graphics primitive
The code verifies if there exists a hyperbolic regular polygon with the given data, checking

\[ A(P) = \pi(s - 2) - s \cdot \alpha > 0, \]

where \( s \) is `sides` and \( \alpha \) is `i_angle`. This raises an error if the `i_angle` is less than the minimum to generate a compact polygon:

```
sage: hyperbolic_regular_polygon(4, pi/2)
Traceback (most recent call last):
  ... ValueError: there exists no hyperbolic regular compact polygon,
  for sides=4 the interior angle must be less than 1/2*pi
```

It is an error to give a center outside the upper half plane in this model:

```
sage: from sage.plot.hyperbolic_regular_polygon import hyperbolic_regular_polygon
sage: hyperbolic_regular_polygon(4, pi/4, 1-I)
Traceback (most recent call last):
  ... ValueError: center: 1.00000000000000 - 1.00000000000000*I is not
    a valid point in the upper half plane model of the hyperbolic plane
```
5.1 Graphics objects

This file contains the definition of the class Graphics. Usually, you don’t call the constructor of this class directly (although you can do it), you would use plot() 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)
- Emily Chen (2013-01-05): Add documentation for show() figsize parameter (trac ticket #5956)
- Eric Gourgoulhon (2015-03-19): Add parameter axes_labels_size (trac ticket #18004)

class sage.plot.graphics.Graphics
    Bases: sage.misc.fast_methods.WithEqualityById, 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.
.. code-block:: sage

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

.. automodule:: sage.repl.rich_output

.. automodule:: sage.plot.add_primitive

.. automodule:: sage.plot.aspect_ratio

---

See :meth:`sage.plot.add_primitive` for details.

EXAMPLES:

.. code-block:: sage

    sage: from sage.plot.add_primitive import add_primitive
    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_label':''})
    sage: A = Arrow(2,-5,.1,.2,{'width':3,'head':0,'rgbcolor':(1,0,0),'linestyle':'dashed','zorder':8,'legend_label':''})
    sage: G.add Primitive(L)
    sage: G.add Primitive(A)
    sage: G
    Graphics object consisting of 2 graphics primitives

---

.. automodule:: sage.plot.aspect_ratio

.. automodule:: sage.plot.aspect_ratio

---

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

.. code-block:: sage

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

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

.. code-block:: sage

    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
   Graphics object consisting of 1 graphics primitive

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
   Graphics object consisting of 1 graphics primitive

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
Graphics object consisting of 1 graphics primitive
```

**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
Graphics object consisting of 1 graphics primitive
```

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

```
sage: plot(sin(x), (x, 0, 10), axes_labels=['x','y'])
Graphics object consisting of 1 graphics primitive
```

**axes_labels_size** *(s=None)*

Set the relative size of axes labels w.r.t. the axes tick marks.

**INPUT:**

• s - float, relative size of axes labels w.r.t. to the tick marks, the size of the tick marks being set by *fontsize()*.
If called with no input, return the current relative size.

EXAMPLES:

```
sage: p = plot(sin(x^2), (x, -3, 3), axes_labels=['$x$', '$y$'])
sage: p.axes_labels_size() # default value
1.6
sage: p.axes_labels_size(2.5)
sage: p.axes_labels_size()  
2.5
```

Now the axes labels are large w.r.t. the tick marks:

```
sage: p  
Graphics object consisting of 1 graphics primitive
```

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

Set the ranges of the $x$ and $y$ axes.

INPUT:

- $\text{xmin}, \text{xmax}, \text{ymin}, \text{ymax}$ - 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 $\text{axes_width}$ setting.

EXAMPLES: 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  
Graphics object consisting of 1 graphics primitive
```

### description()

Print a textual description to stdout.

This method is mostly used for doctests.

EXAMPLES:
```
sage: print(polytopes.hypercube(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)]
```

**flip** *(flip_x=False, flip_y=False)*

Get the flip options and optionally mirror this graphics object.

**INPUT:**

- `flip_x` – boolean (default: False); if True, replace the current `flip_x` option by its opposite
- `flip_y` – boolean (default: False); if True, replace the current `flip_y` option by its opposite

**OUTPUT:** a tuple containing the new flip options

**EXAMPLES:**

When called without arguments, this just returns the current flip options:

```
sage: L = line([(1, 0), (2, 3)])
sage: L.flip()
(False, False)
```

Otherwise, the specified options are changed and the new options are returned:

```
sage: L.flip(flip_y=True)
(False, True)
sage: L.flip(True, True)
(True, False)
```

**fontsize** *(s=None)*

Set the font size of axes labels and tick marks.

Note that the relative size of the axes labels font w.r.t. the tick marks font can be adjusted via `axes_labels_size()`.

**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
Graphics object consisting of 1 graphics primitive
```
**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:**

```python
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 the x and y coordinate minimum and maximum.

**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()`.

**OUTPUT:**

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

**EXAMPLES:**

```python
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()`:

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

The width/height ratio (in output units, after factoring in the chosen aspect ratio) of the plot is limited to $10^{-15} \ldots 10^{15}$, otherwise floating point errors cause problems in matplotlib:

```python
sage: l = line([(1e-19,-1), (-1e-19,+1)], aspect_ratio=1.0)
sage: l.get_minmax_data()
{'xmax': 1.00010000000000e-15, 'xmin': -9.99900000000000e-16, 'ymax': 1.0, 'ymin': -1.0}
sage: l = line([(0,0), (1,1)], aspect_ratio=1e19)
sage: l.get_minmax_data()
{'xmax': 5000.50000000000, 'xmin': -4999.50000000000, 'ymax': 1.0, 'ymin': 0.0}
```
inset (graphics, pos=None, fontsize=None)
Add a graphics object as an inset.

INPUT:

- graphics – the graphics object (instance of Graphics) to be added as an inset to the current graphics
- pos – (default: None) 4-tuple (left, bottom, width, height) specifying the location and size of the inset on the final figure, all quantities being in fractions of the figure width and height; if None, the value (0.7, 0.7, 0.2, 0.2) is used
- fontsize – (default: None) integer, font size (in points) for the inset; if None, the value of 6 points is used, unless fontsize has been explicitly set in the construction of graphics (in this case, it is not overwritten here)

OUTPUT:

- instance of MultiGraphics

EXAMPLES:

```sage
sage: f(x) = x^2*sin(1/x)
sage: g1 = plot(f(x), (x, -2, 2), axes_labels=['$x$', '$y$'])
sage: g2 = plot(f(x), (x, -0.3, 0.3), axes_labels=['$x$', '$y$'],
....:            frame=True)
sage: g1.inset(g2)
Multigraphics with 2 elements
```

```
  x
-2.0 -1.5 -1.0 -0.5 0.5 1.0 1.5 2.0
  y
-1.5 -1.0 -0.5 0.0 0.5 1.0 1.5
  0.0 0.1 0.2 0.3
  0.04 0.02 0.00
```

```sage
sage: f(x) = x^2*sin(1/x)
```
Using non-default values for the position/size and the font size:

```sage
g1.inset(g2, pos=(0.15, 0.7, 0.25, 0.25), fontsize=8)
```

Multigraphics with 2 elements

We can add another inset by invoking `inset` on the last output:

```sage
g1g2 =
g3 = plot(f(x), (x, -0.05, 0.05), axes_labels=['$x$', '$y$'],
.....:    frame=True)
g1g2.inset(g3, pos=(0.65, 0.12, 0.25, 0.25))
```

Multigraphics with 3 elements

**legend** *(show=\texttt{None})*

Set whether or not the legend is shown by default.

**INPUT:**

- **show** *(default: \texttt{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
Graphics object consisting of 1 graphics primitive

matplotlib (filename=None, xmin=None, xmax=None, ymin=None, ymax=None, figsize=None, figure=None, sub=None, axes=None, axes_labels=None, axes_labels_size=None, flip_x=False, flip_y=False, 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, stylesheet=None, typeset='default')

Construct or modify a Matplotlib figure by drawing self on it.

INPUT (partial description, involving only Matplotlib objects; see show() for the other arguments):

• figure – (default: None) Matplotlib figure (class matplotlib.figure.Figure) on which self is to be displayed; if None, the figure will be created from the parameter figsize

• figsize – (default: None) width or [width, height] in inches of the Matplotlib figure in case figure is None; if figsize is None, Matplotlib's default (6.4 x 4.8 inches) is used

• sub – (default: None) subpart of the figure, as an instance of Matplotlib “axes” (class matplotlib.axes.Axes) on which self is to be drawn; if None, the subpart will be created so as to cover the whole figure

OUTPUT:

• a matplotlib.figure.Figure object; if the argument figure is provided, this is the same object as figure.

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]

plot()

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
```

It does not accept any argument (trac ticket #19539):

```
sage: S.plot(1)  # py2
Traceback (most recent call last):
   ...
TypeError: plot() takes exactly 1 argument (2 given)
sage: S.plot(1)  # py3
Traceback (most recent call last):
   ...
TypeError: plot() takes 1 positional argument but 2 were given
sage: S.plot(hey="hou")
Traceback (most recent call last):
   ...
TypeError: plot() got an unexpected keyword argument 'hey'
```

```
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
Graphics3d Object
```
```
save(filename, legend_back_color='white', legend_borderpad=0.6, legend_borderaxespad=None, legend_columnspacing=None, legend_fancybox=False, legend_font_family='sans-serif', legend_font_size='medium', legend_font_style='normal', legend_font_variant='normal', legend_font_weight='medium', legend_HANDLElength=0.05, legend_HANDLEtextpad=0.5, legend_labelspacing=0.02, legend_loc='best', legend_markerscale=0.6, legend_ncol=1, legend númpoints=2, legend_shadow=True, legend_title=None, **kwds)
Save the graphics to an image file.
INPUT:
- filename – string. The filename and the image format given by the extension, which can be one of the following:
  - .eps,
  - .pdf,
  - .pgf,
  - .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)
```

Graphics object consisting of 1 graphics primitive

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

**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
Graphics object consisting of 1 graphics primitive
```

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
Graphics object consisting of 1 graphics primitive
```

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))
Graphics object consisting of 2 graphics primitives
```

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
Graphics object consisting of 2 graphics primitives
sage: (Q + P).aspect_ratio(); Q+P
2.0
Graphics object consisting of 2 graphics primitives
```

`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_flip(flip_x=None, flip_y=None)`

Set the flip options for this graphics object.

**INPUT:**
- $flip_x$ - boolean (default: None); if not None, set the $flip_x$ option to this value
- $flip_y$ - boolean (default: None); if not None, set the $flip_y$ option to this value

**EXAMPLES:**

```
sage: L = line([(1, 0), (2, 3)])
sage: L.set_flip(flip_y=True)
sage: L.flip()
(False, True)
```

(continues on next page)
 sage: L.set_flip(True, False)
sage: L.flip()
(True, False)

**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: ‘white’) 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_style - (default: ‘normal’) string, one of ‘normal’, ‘italic’, ‘oblique’
• font_variant - (default: ‘normal’) string, one of ‘normal’, ‘small-caps’
• 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: True) 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 without a shadow:

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

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
Graphics object consisting of 1 graphics primitive

sage: p.set_legend_options(loc=10); p
Graphics object consisting of 1 graphics primitive

sage: p.set_legend_options(loc=(0.5,0.5)); p # aligns the bottom of the box to the center
Graphics object consisting of 1 graphics primitive
```

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

Show this graphics image immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

OPTIONAL INPUT:

• dpi - (default: 100) dots per inch
• **figsize** - (default: [6.4, 4.8]) [width, height] inches. The maximum value of each of the width and the height can be 327 inches, at the default dpi of 100 dpi, which is just shy of the maximum allowed value of 32768 dots (pixels).

• **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.

• **axes_labels_size** - (default: current setting – 1.6) scale factor relating the size of the axes labels with respect to the size of the tick marks.

• **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.

• **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 \( \pi \), \( e \), 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.

- **xmin** – starting x value in the rendered figure.
- **xmax** – ending x value in the rendered figure.
- **ymin** – starting y value in the rendered figure.
- **ymax** – ending y value in the rendered figure.
- **flip_x** – (default: False) boolean. If True, flip the horizontal axis.
- **flip_y** – (default: False) boolean. If True, flip the vertical axis.
- **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 https://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.

**OUTPUT:**
This method does not return anything. Use `save()` if you want to save the figure as an image.

**EXAMPLES:**

```python
sage: c = circle((1,1), 1, color='red')
sage: c.show(xmin=-1, xmax=3, ymin=-1, ymax=3)
```
You can make the picture larger by changing `figsize` with width, height each having a maximum value of 327 inches at default dpi:

```
sage: p = ellipse((0,0),4,1)
sage: p.show(figsize=[327,10],dpi=100)
sage: p.show(figsize=[328,10],dpi=80)
```

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=r'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: p = 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)
```

Prior to trac ticket #19485, legends by default had a shadowless gray background. This behavior can be recovered by passing in certain `legend_options`:

```
sage: p = plot(sin(x), legend_label=r'$\sin(x)$')
sage: p.show(legend_options={'back_color': (0.9,0.9,0.9), 'shadow': False})
```

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.

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([10**i for i in 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)
```

Flip the horizontal or vertical axis.

```
sage: G = plot(x^3, -2, 3)
sage: G.show(flip_x=True)
sage: G.show(flip_y=True)
```

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.
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], range(-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":"-"}))
.....: )
.....: )
.....: gridlinesstyle=dict(marker='x',color="black"))
```

Grid lines can be added to contour plots.
Grid lines can be added to matrix plots.

```python
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:

```python
sage: plot(sin(x), (x, -pi, pi),thickness=2)+point((pi, -1), pointsize=15)
Graphics object consisting of 2 graphics primitives
sage: plot(sin(x), (x, -pi, pi),thickness=2,axes_pad=0)+point((pi, -1),
    pointsize=15)
Graphics object consisting of 2 graphics primitives
```

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 $1/b^{\text{axes}_\text{pad}}$ 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.

```python
sage: plot_loglog(x, (1.1*10**-2, 9990))
Graphics object consisting of 1 graphics primitive
sage: plot_loglog(x, (1.1*10**-2, 9990), axes_pad=0)
Graphics object consisting of 1 graphics primitive
```

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

Here the labels are not so useful:

```python
sage: plot(sin(pi*x), (x, -8, 8))
Graphics object consisting of 1 graphics primitive
```

Now put ticks at multiples of 2:

```python
sage: plot(sin(pi*x), (x, -8, 8), ticks=2)
Graphics object consisting of 1 graphics primitive
```

Or just choose where you want the ticks:

```python
sage: plot(sin(pi*x), (x, -8, 8), ticks=[[-7,-3,0,3,7],[-1/2,0,1/2]])
Graphics object consisting of 1 graphics primitive
```

Or no ticks at all:

```python
sage: plot(sin(pi*x), (x, -8, 8), ticks=[[],[]])
Graphics object consisting of 1 graphics primitive
```

This can be very helpful in showing certain features of plots.
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`).
```
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.
Graphics object consisting of 1 graphics primitive
```

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]])
Graphics object consisting of 1 graphics primitive
```

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)
Graphics object consisting of 1 graphics primitive
```

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

```
sage: plot(x, -4, 4, title='Plot x', title_pos=0.05)
doctest:...: ...RichReprWarning: Exception in __rich_repr__ while displaying...
object: 'title_pos' must be a list or tuple of two real numbers.
Graphics object consisting of 1 graphics primitive
```

**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
Graphics object consisting of 1 graphics primitive
```

**xmax** *(xmax=None)*

**EXAMPLES:**

5.1. Graphics objects

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

\texttt{xmin}(\texttt{xmin=None})
\textbf{EXAMPLES:}

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

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

\texttt{ymax}(\texttt{ymax=None})
\textbf{EXAMPLES:}

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

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

\texttt{ymin}(\texttt{ymin=None})
\textbf{EXAMPLES:}

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

\texttt{sage.plot.graphics.GraphicsArray}(\texttt{*args, **kwargs})
This is deprecated (see trac ticket #28675). Use \texttt{sage.plot.multigraphics.GraphicsArray} instead.

\texttt{sage.plot.graphics.is_Graphics}(x)
Return True if \texttt{x} is a Graphics object.

\textbf{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
```
5.2 Graphics arrays and insets

This module defines the classes `MultiGraphics` and `GraphicsArray`. The class `MultiGraphics` is the base class for 2-dimensional graphical objects that are composed of various `Graphics` objects, arranged in a given canvas. The subclass `GraphicsArray` is for `Graphics` objects arranged in a regular array.

AUTHORS:

- Eric Gourgoulhon (2019-05-24): initial version, refactoring the class `GraphicsArray` that was defined in the module `graphics`.

```python
class sage.plot.multigraphics.GraphicsArray(array)
Bases: sage.plot.multigraphics.MultiGraphics

This class implements 2-dimensional graphical objects that constitute an array of `Graphics` drawn on a single canvas.

The user interface is through the function `graphics_array()`.

INPUT:

- `array` – either a list of lists of `Graphics` elements (generic case) or a single list of `Graphics` elements (case of a single-row array)

EXAMPLES:

An array made of four graphics objects:

```python
sage: g1 = plot(sin(x^2), (x, 0, 6), axes_labels=['$x$', '$y$'],
.....:     axes=False, frame=True, gridlines='minor')
sage: y = var('y')
sage: g2 = streamline_plot((sin(x), cos(y)), (x,-3,3), (y,-3,3),
.....:     aspect_ratio=1)
sage: g3 = graphs.DodecahedralGraph().plot()
sage: g4 = polar_plot(sin(5*x)^2, (x, 0, 2*pi), color='green',
.....:     fontsize=8) \
.....:     + circle((0,0), 0.5, rgbcolor='red', fill=True, alpha=0.1,
.....:     legend_label='pink')
sage: g4.set_legend_options(loc='upper right')
sage: G = graphics_array([[g1, g2], [g3, g4]])
sage: G
Graphics Array of size 2 x 2
```

If one constructs the graphics array from a single list of graphics objects, one obtains a single-row array:

```python
sage: G = graphics_array([g1, g2, g3, g4])
sage: G
Graphics Array of size 1 x 4
```

We note that the overall aspect ratio of the figure is 4/3 (the default), which makes `g1` elongated, while the aspect ratio of `g2`, which has been specified with the parameter `aspect_ratio=1` is preserved. To get a better aspect ratio for the whole figure, one can use the option `figsize` in the method `show()`:

```python
sage: G.show(figsize=[8, 3])
```

We can access individual elements of the graphics array with the square-bracket operator:

```python
sage: G = graphics_array([[g1, g2], [g3, g4]])  # back to the 2x2 array
sage: print(G)
```

(continues on next page)
5.2. Graphics arrays and insets
Graphics Array of size 2 x 2

```
sage: G[0] is g1
True
sage: G[1] is g2
True
sage: G[2] is g3
True
sage: G[3] is g4
True
```

Note that with respect to the square-bracket operator, $G$ is considered as a flattened list of graphics objects, not as an array. For instance, $G[0, 1]$ throws an error:

```
sage: G[0, 1]  # py3 (error message is slightly different with Python 2)
Traceback (most recent call last):
...TypeError: list indices must be integers or slices, not tuple
```

$G[:]$ returns the full (flattened) list of graphics objects composing $G$:

```
sage: G[:]
[Graphics object consisting of 1 graphics primitive,
 Graphics object consisting of 1 graphics primitive,
 Graphics object consisting of 51 graphics primitives,
 Graphics object consisting of 2 graphics primitives]
```

The total number of Graphics objects composing the array is returned by the function `len`:

```
sage: len(G)
4
```

The square-bracket operator can be used to replace elements in the array:

```
sage: G[0] = g4
sage: G
Graphics Array of size 2 x 2
```

append ($g$)

Append a graphics to the array.

Currently not 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.
5.2. Graphics arrays and insets
EXAMPLES:

```python
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
```

```
position(index)

Return the position and relative size of an element of self on the canvas.

INPUT:

• index – integer specifying which element of self

OUTPUT:

• a 4-tuple (left, bottom, width, height) giving the location and relative size of the element on the canvas, all quantities being expressed in fractions of the canvas width and height

EXAMPLES:
```n
sage: g1 = plot(sin(x), (x, -pi, pi))
sage: g2 = circle((0,1), 1.)
sage: G = graphics_array([g1, g2])
sage: G.position(0)  # tol 5.0e-3
(0.025045451349937315, 0.03415488992713045, 0.4489880779745068, 0.9345951100728696)
sage: G.position(1)  # tol 5.0e-3
(0.5170637412999687, 0.20212705964722733, 0.4489880779745068, 0.5986507706326758)
```n
```
class sage.plot.multigraphics.MultiGraphics(graphics_list)

Bases: sage.misc.fast_methods.WithEqualityById, sage.structure.sage_object.SageObject

Base class for objects composed of Graphics objects.

Both the display and the output to a file of MultiGraphics objects are governed by the method save(), which is called by the rich output display manager, via graphics_from_save().

The user interface is through the functions multi_graphics() (generic multi-graphics) and graphics_array() (subclass GraphicsArray).

INPUT:

• graphics_list – a list of graphics along with their positions on the common canvas; each element of graphics_list is either
  - a pair (graphics, position), where graphics is a Graphics object and position is the 4-tuple (left, bottom, width, height) specifying the location and size of the graphics on the canvas, all quantities being in fractions of the canvas width and height
  - or a single Graphics object; its position is then assumed to occupy the whole canvas, except for some padding; this corresponds to the default position (left, bottom, width, height) = (0.125, 0.11, 0.775, 0.77)
EXAMPLES:

A multi-graphics made from two graphics objects:

```
sage: g1 = plot(sin(x^3), (x, -pi, pi))
sage: g2 = circle((0,0), 1, color='red')
sage: G = multi_graphics([(g1, (0.2, 0.55, 0.3, 0.3))])
sage: G
Multigraphics with 2 elements
```

Since no position was given for `g1`, it occupies the whole canvas. Moreover, we note that `g2` has been drawn over `g1` with a white background. To have a transparent background instead, one has to construct `g2` with the keyword `transparent` set to `True`:

```
sage: g2 = circle((0,0), 1, color='red', transparent=True)
sage: G = multi_graphics([(g1, (0.2, 0.55, 0.3, 0.3))])
sage: G
Multigraphics with 2 elements
```

We can add a new graphics object to `G` via the method `append()`:

```
sage: g3 = complex_plot(zeta, (-20, 10), (-20, 20),
...: axes_labels='$x$', '$y$')
.. frame=True)
sage: G.append(g3, pos=(0.63, 0.12, 0.3, 0.3))
sage: G
Multigraphics with 3 elements
```

5.2. Graphics arrays and insets
5.2. Graphics arrays and insets
We can access the individual elements composing \( G \) with the square-bracket operator:

```
sage: print(G[0])
Graphics object consisting of 1 graphics primitive
sage: G[0] is g1
True
sage: G[1] is g2
True
sage: G[2] is g3
True
```

\( G[:] \) returns the full list of graphics objects composing \( G \):

```
sage: G[:]
[Graphics object consisting of 1 graphics primitive,
 Graphics object consisting of 1 graphics primitive,
 Graphics object consisting of 1 graphics primitive]
sage: len(G)
3
```

`append(graphics, pos=None)`

Append a graphics object to \( self \).

**INPUT:**

- `graphics` – the graphics object (instance of `Graphics`) to be added to \( self \)
- `pos` – (default: None) 4-tuple \((\text{left, bottom, width, height})\) specifying the location and size of `graphics` on the canvas, all quantities being in fractions of the canvas width and height; if None, `graphics` is assumed to occupy the whole canvas, except for some padding; this corresponds to the default position \((\text{left, bottom, width, height}) = (0.125, 0.11, 0.775, 0.77)\)

**EXAMPLES:**

Let us consider a multigraphics with 2 elements:

```
sage: g1 = plot(chebyshev_T(4, x), (x, -1, 1), title='n=4')
sage: g2 = plot(chebyshev_T(8, x), (x, -1, 1), title='n=8',
                 ....:     color='red')
sage: G = multi_graphics(
                 ....:     [(g1, (0.125, 0.2, 0.4, 0.4)),
                 ....:      (g2, (0.55, 0.4, 0.4, 0.4))])
sage: G
Multigraphics with 2 elements
```

We append a third plot to it:

```
sage: g3 = plot(chebyshev_T(16, x), (x, -1, 1), title='n=16',
                 ....:     color='brown')
sage: G.append(g3, pos=(0.55, 0.11, 0.4, 0.15))
sage: G
Multigraphics with 3 elements
```

We may use `append` to add a title:

```
sage: title = text("Chebyshev polynomials", (0, 0), fontsize=16,
                 ....:     axes=False)
sage: G.append(title, pos=(0.18, 0.8, 0.7, 0.1))
sage: G
Multigraphics with 4 elements
```
5.2. Graphics arrays and insets

- Two graphics side by side, one labeled "n=4" and the other labeled "n=8".

- The "n=4" graph shows a sine wave with a blue line.
- The "n=8" graph shows a sine wave with a red line.

- Both graphs are set within a coordinate system ranging from -1.0 to 1.0 on both axes.

- The y-axis ranges from -1.0 to 1.0.

- The graphs illustrate how the number of periods in a sine wave can affect its appearance.
Chebyshev polynomials

\begin{align*}
n=4 & \quad & n=8 & \quad & n=16 \\
\end{align*}

5.2. Graphics arrays and insets
See also:

```
inset()
```

```
inset(graphics, pos=None, fontsize=None)
```
Add a graphics object as an inset.

**INPUT:**

- `graphics` – the graphics object (instance of `Graphics`) to be added as an inset
- `pos` – (default: None) 4-tuple (left, bottom, width, height) specifying the location and relative size of the inset on the canvas, all quantities being expressed in fractions of the canvas width and height; if None, the value (0.7, 0.7, 0.2, 0.2) is used
- `fontsize` – (default: None) integer, font size (in points) for the inset; if None, the value of 6 points is used, unless fontsize has been explicitly set in the construction of graphics (in this case, it is not overwritten here)

**OUTPUT:**

- instance of `MultiGraphics`

**EXAMPLES:**

Let us consider a graphics array of 2 elements:

```
sage: G = graphics_array([plot(sin, (0, 2*pi)), ....: plot(cos, (0, 2*pi))])
sage: G
Graphics Array of size 1 x 2
```

and add some inset at the default position:

```
sage: c = circle((0,0), 1, color='red', thickness=2, frame=True)
sage: G.inset(c)
Multigraphics with 3 elements
```

We may customize the position and font size of the inset:

```
sage: G.inset(c, pos=(0.3, 0.7, 0.2, 0.2), fontsize=8)
Multigraphics with 3 elements
```

```
matplotlib(figure=None, figsize=None, **kwds)
```
Construct or modify a Matplotlib figure by drawing `self` on it.

**INPUT:**

- `figure` – (default: None) Matplotlib figure (class `matplotlib.figure.Figure`) on which `self` is to be displayed; if None, the figure will be created from the parameter figsize
- `figsize` – (default: None) width or [width, height] in inches of the Matplotlib figure in case figure is None; if figsize is None, Matplotlib’s default (6.4 x 4.8 inches) is used
- `kwds` – options passed to the `matplotlib()` method of each graphics object constituting `self`

**OUTPUT:**

- a `matplotlib.figure.Figure` object; if the argument `figure` is provided, this is the same object as `figure`. 

**EXAMPLES:**

Let us consider a `GraphicsArray` object with 3 elements:
5.2. Graphics arrays and insets
5.2. Graphics arrays and insets
plot()  
Return self since self is already a graphics object.

EXAMPLES:
sage: g1 = plot(cos, 0, 1)
sage: g2 = circle((0,0), 1)
sage: G = multi_graphics([g1, g2])
sage: G.plot() is G
True

position(index)

Return the position and relative size of an element of self on the canvas.

INPUT:

• index – integer specifying which element of self

OUTPUT:

• a 4-tuple (left, bottom, width, height) giving the location and relative size of the element on the canvas, all quantities being expressed in fractions of the canvas width and height

EXAMPLES:

sage: g1 = plot(sin(x^2), (x, 0, 4))
sage: g2 = circle((0,0), 1, rgbcolor='red', fill=True, axes=False)
sage: G = multi_graphics([g1, (g2, (0.15, 0.2, 0.1, 0.15))])
sage: G.position(0)  # tol 1.0e-13
(0.125, 0.11, 0.775, 0.77)
sage: G.position(1)  # tol 1.0e-13
(0.15, 0.2, 0.1, 0.15)

save(filename, figsize=None, **kwds)

Save self to a file, in various formats.

INPUT:

• filename – (string) the file name; the image format is 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.

• figsize – (default: None) width or [width, height] in inches of the Matplotlib figure; if none is provided, Matplotlib’s default (6.4 x 4.8 inches) is used

• kwds – keyword arguments, like dpi=..., passed to the plotter, see show()

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)
**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:

```python
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(ext='.png'))
```

**show** *(**kwds)***

Show `self` immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

**OPTIONAL INPUT:**

- **dpi** – dots per inch
- **figsize** – width or [width, height] of the figure, in inches; the default is 6.4 x 4.8 inches
- **axes** – boolean; if True, all individual graphics are endowed with axes; if False, all axes are removed (this overrides the axes option set in each graphics)
- **frame** – boolean; if True, all individual graphics are drawn with a frame around them; if False, all frames are removed (this overrides the frame option set in each graphics)
- **fontsize** – positive integer, the size of fonts for the axes labels (this overrides the fontsize option set in each graphics)

**OUTPUT:**

This method does not return anything. Use `save()` if you want to save the figure as an image.

### EXAMPLES:

This draws a graphics array with four trig plots and no axes in any of the plots and a figure width of 4 inches:

```python
sage: G = graphics_array([[[plot(sin), plot(cos)],
.....:   [plot(tan), plot(sec)]]])
sage: G.show(axes=False, figsize=4)
```

Same thing with a frame around each individual graphics:

```python
sage: G.show(axes=False, frame=True, figsize=4)
```

Actually, many options are possible; for instance, we may set `fontsize` and `gridlines`:

```python
sage: G.show(axes=False, frame=True, figsize=4, fontsize=8,
.....:   gridlines='major')
```
5.3 Plotting primitives

**class** `sage.plot.primitive.GraphicPrimitive` *(options)*

Bases: `sage.misc.fast_methods.WithEqualityById, 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:

```python
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:**

```python
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:**

```python
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:

```python
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:

```python
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
Graphics object consisting of 2 graphics primitives
sage: q=Q[0]
sage: q.set_zorder(3)
sage: P+Q
# teal line on top
Graphics object consisting of 2 graphics primitives
sage: q.options()['zorder']
3
```

**class** `sage.plot.primitive.GraphicPrimitive_xydata` (*options*)

Bases: `sage.plot.primitive.GraphicPrimitive`

**get_minmax_data** ()

Returns a dictionary with the bounding box data.

EXAMPLES:

```python
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
```

```python
sage: d = point((3, 3), rgbcolor=hue(0.75))[0].get_minmax_data()
sage: d['xmin']
3.0
sage: d['ymin']
3.0
```

5.3. Plotting primitives
5.4 Plotting utilities

`sage.plot.misc.get_matplotlib_linestyle(linestyle, return_type)`

Function which translates between matplotlib linestyle in short notation (i.e. ‘-’, ‘--’, ‘.’, ‘-.’) and long notation (i.e. ‘solid’, ‘dashed’, ‘dotted’, ‘dashdot’).

If linestyle is none of these allowed options, the function raises a ValueError.

**INPUT:**

- **linestyle** - 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)

  If linestyle is None (of type NoneType), then we return it back unmodified.

- **return_type** - The type of linestyle that should be output. This argument takes only two values - "long" or "short".

**EXAMPLES:**

Here is an example how to call this function:

```sage
sage: from sage.plot.misc import get_matplotlib_linestyle
sage: get_matplotlib_linestyle(':', return_type='short')
':'
sage: get_matplotlib_linestyle(':', return_type='long')
'dotted'
```

`sage.plot.misc.setup_for_eval_on_grid(funcs, ranges, plot_points=None, return_vars=False)`

Calculate the necessary parameters to construct a list of points, and make the functions fast_callable.

**INPUT:**

- **funcs** – a function, or a list, tuple, or vector of functions
• **ranges** – a list of ranges. A range can be a 2-tuple of numbers specifying the minimum and maximum, or a 3-tuple giving the variable explicitly.

• **plot_points** – a tuple of integers specifying the number of plot points for each range. If a single number is specified, it will be the value for all ranges. This defaults to 2.

• **return_vars** – (default False) If True, return the variables, in order.

**OUTPUT:**

• **fast_funcs** - if only one function passed, then a fast callable function. If funcs is a list or tuple, then a tuple of fast callable functions is returned.

• **range_specs** - a list of range_specs: for each range, a tuple is returned of the form (range_min, range_max, range_step) such that 
\[ \text{srange}(\text{range_min}, \text{range_max}, \text{range_step}) \]
gives the correct points for evaluation.

**EXAMPLES:**

```python
sage: x,y,z=var('x,y,z')
sage: f(x,y)=x+y-z
sage: g(x,y)=x+y
sage: h(y)=-y
sage: sage.plot.misc.setup_for_eval_on_grid(f, [(0, 2),(1,3),(-4,1)], plot_points=5)
((<sage.ext...>, [(0.0, 2.0, 0.5), (1.0, 3.0, 0.5), (-4.0, 1.0, 1.25)]))
sage: sage.plot.misc.setup_for_eval_on_grid([g,h], [(0, 2),(-1,1)], plot_points=5)
((<sage.ext...>, <sage.ext...>), [(0.0, 2.0, 0.5), (-1.0, 1.0, 0.5)])
sage: sage.plot.misc.setup_for_eval_on_grid([sin,cos], [(-1,1)], plot_points=9)
((<sage.ext...>, <sage.ext...>), [(-1.0, 1.0, 0.25)])
sage: sage.plot.misc.setup_for_eval_on_grid([lambda x: x^2,cos], [(-1,1)], plot_points=9)
((<function <lambda> ...>, <sage.ext...>), [(-1.0, 1.0, 0.25)])
```

Beware typos: a comma which should be a period, for instance:

```python
sage: sage.plot.misc.setup_for_eval_on_grid(x+y, [(x, 1, 2), (y, 0,1, 0.2)], plot_points=[4,9,10])
Traceback (most recent call last):
... ValueError: plot_points must be either an integer or a list of integers, one for each range
```

Beware typos: a comma which should be a period, for instance:

```python
sage: sage.plot.misc.setup_for_eval_on_grid(x+y, [(x, 1, 2), (y, 0,1, 0.2)], plot_points=[4,9,10])
Traceback (most recent call last):
... ValueError: At least one variable range has more than 3 entries: each should either have 2 or 3 entries, with one of the forms (xmin, xmax) or (x, xmin, xmax)
```

(continues on next page)
sage.plot.misc.unify_arguments(funcs)

Return a tuple of variables of the functions, as well as the number of “free” variables (i.e., variables that defined in a callable function).

INPUT:
- funcs – a list of functions; these can be symbolic expressions, polynomials, etc

OUTPUT: functions, expected arguments
- A tuple of variables in the functions
- A tuple of variables that were “free” in the functions

EXAMPLES:

```
sage: x, y, z = var('x, y, z')
sage: f(x, y) = x + y + z
sage: g(x, y) = x + y
sage: h(y) = -y
sage: sage.plot.misc.unify_arguments((f, g, h))
((x, y, z), (z))
```

```
sage: sage.plot.misc.unify_arguments((f, g, h))
((x, y, z), (z))
```

```
sage: sage.plot.misc.unify_arguments((x+y, x-y))
((x, y), (x, y))
```
CHAPTER

SIX

INDICES AND TABLES

• Index
• Module Index
• Search Page
PYTHON MODULE INDEX

g
sage.graphs.graph_plot, 285

p
sage.plot.animate, 165
sage.plot.arc, 323
sage.plot.arrow, 325
sage.plot.bar_chart, 283
sage.plot.bezier_path, 338
sage.plot.circle, 344
sage.plot.colors, 155
sage.plot.complex_plot, 177
sage.plot.contour_plot, 185
sage.plot.density_plot, 252
sage.plot.disk, 356
sage.plot.ellipse, 358
sage.plot.graphics, 397
sage.plot.histogram, 280
sage.plot.hyperbolic_arc, 385
sage.plot.hyperbolic_polygon, 387
sage.plot.hyperbolic_regular_polygon, 391
sage.plot.line, 360
sage.plot.matrix_plot, 316
sage.plot.misc, 446
sage.plot.multigraphics, 423
sage.plot.plot, 1
sage.plot.plot_field, 260
sage.plot.point, 364
sage.plot.polygon, 369
sage.plot.primitive, 444
sage.plot.scatter_plot, 278
sage.plot.step, 280
sage.plot.streamline_plot, 271
sage.plot.text, 143
INDEX

Symbols

_rich_repr_() (sage.plot.graphics.Graphics method), 398

A

adaptive_refinement () (in module sage.plot.plot), 28
add_frame () (sage.plot.animate.APngAssembler method), 167
add_primitive () (sage.plot.graphics.Graphics method), 398
animate () (in module sage.plot.animate), 175
Animation (class in sage.plot.animate), 168
apng () (sage.plot.animate.Animation method), 169
APngAssembler (class in sage.plot.animate), 167
append () (sage.plot.multigraphics.GraphicsArray method), 426
append () (sage.plot.multigraphics.MultiGraphics method), 432
Arc (class in sage.plot.arc), 323
arc () (in module sage.plot.arc), 324
Arrow (class in sage.plot.arrow), 324
arrow () (in module sage.plot.arrow), 326
arrow2d () (in module sage.plot.arrow), 327
aspect_ratio () (sage.plot.graphics.Graphics method), 398
axes () (sage.plot.graphics.Graphics method), 398
axes_color () (sage.plot.graphics.Graphics method), 399
axes_label_color () (sage.plot.graphics.Graphics method), 399
axes_labels () (sage.plot.graphics.Graphics method), 400
axes_labels_size () (sage.plot.graphics.Graphics method), 400
axes_range () (sage.plot.graphics.Graphics method), 401
axes_width () (sage.plot.graphics.Graphics method), 401

B

bar_chart () (in module sage.plot.bar_chart), 283
BarChart (class in sage.plot.bar_chart), 283
bezier_path () (in module sage.plot.bezier_path), 340
bezier_path () (sage.plot.arc.Arc method), 323
BezierPath (class in sage.plot.bezier_path), 338
blend () (sage.plot.colors.Color method), 155

C

check_color_data () (in module sage.plot.colors), 159
Circle (class in sage.plot.circle), 344

circle() (in module sage.plot.circle), 345

Color (class in sage.plot.colors), 155

Colormaps (class in sage.plot.colors), 158

ColorsDict (class in sage.plot.colors), 159

complex_plot() (in module sage.plot.complex_plot), 177

complex_to_rgb() (in module sage.plot.complex_plot), 183

ComplexPlot (class in sage.plot.complex_plot), 177

contour_plot() (in module sage.plot.contour_plot), 185

ContourPlot (class in sage.plot.contour_plot), 185

CurveArrow (class in sage.plot.arrow), 326

D
darker() (sage.plot.colors.Color method), 156

density_plot() (in module sage.plot.density_plot), 252

DensityPlot (class in sage.plot.density_plot), 252

description() (sage.plot.graphics.Graphics method), 401

Disk (class in sage.plot.disk), 356

disk() (in module sage.plot.disk), 357

E

Ellipse (class in sage.plot.ellipse), 358

equify() (in module sage.plot.contour_plot), 220

F

ffmpeg() (sage.plot.animate.Animation method), 169

flip() (sage.plot.graphics.Graphics method), 402

float_to_html() (in module sage.plot.colors), 159

float_to_integer() (in module sage.plot.colors), 160

fontsize() (sage.plot.graphics.Graphics method), 402

G

generate_plot_points() (in module sage.plot.plot), 28

generate_plot_points() (sage.plot.graphics.Graphics method), 402

circle() (sage.plot.colors.Color method), 156

circle() (in module sage.plot.circle), 345

Color (class in sage.plot.colors), 155

Colormaps (class in sage.plot.colors), 158

ColorsDict (class in sage.plot.colors), 159

complex_plot() (in module sage.plot.complex_plot), 177

complex_to_rgb() (in module sage.plot.complex_plot), 183

ComplexPlot (class in sage.plot.complex_plot), 177

contour_plot() (in module sage.plot.contour_plot), 185

ContourPlot (class in sage.plot.contour_plot), 185

CurveArrow (class in sage.plot.arrow), 326

D
darker() (sage.plot.colors.Color method), 156

density_plot() (in module sage.plot.density_plot), 252

DensityPlot (class in sage.plot.density_plot), 252

description() (sage.plot.graphics.Graphics method), 401

Disk (class in sage.plot.disk), 356

disk() (in module sage.plot.disk), 357

E

Ellipse (class in sage.plot.ellipse), 358

equify() (in module sage.plot.contour_plot), 220

F

ffmpeg() (sage.plot.animate.Animation method), 169

flip() (sage.plot.graphics.Graphics method), 402

float_to_html() (in module sage.plot.colors), 159

float_to_integer() (in module sage.plot.colors), 160

fontsize() (sage.plot.graphics.Graphics method), 402

G

generate_plot_points() (in module sage.plot.plot), 28

generate_plot_points() (sage.plot.graphics.Graphics method), 402

circle() (sage.plot.colors.Color method), 156

circle() (in module sage.plot.circle), 345

Color (class in sage.plot.colors), 155

Colormaps (class in sage.plot.colors), 158

ColorsDict (class in sage.plot.colors), 159

complex_plot() (in module sage.plot.complex_plot), 177

complex_to_rgb() (in module sage.plot.complex_plot), 183

ComplexPlot (class in sage.plot.complex_plot), 177

contour_plot() (in module sage.plot.contour_plot), 185

ContourPlot (class in sage.plot.contour_plot), 185

CurveArrow (class in sage.plot.arrow), 326
get_minmax_data() (sage.plot.matrix_plot.MatrixPlot method), 316
get_minmax_data() (sage.plot.plot_field.PlotField method), 260
get_minmax_data() (sage.plot.primitive.GraphicPrimitive_xydata method), 445
get_minmax_data() (sage.plot.scatter_plot.ScatterPlot method), 279
get_minmax_data() (sage.plot.streamline_plot.StreamlinePlot method), 271
get_minmax_data() (sage.plot.text.Text method), 143
gif() (sage.plot.animate.Animation method), 170
GraphicPrimitive (class in sage.plot.primitive), 444
GraphicPrimitive_xydata (class in sage.plot.primitive), 445
Graphics (class in sage.plot.graphics), 397
graphics_array() (in module sage.plot.plot), 29
graphics_array() (sage.plot.animate.Animation method), 171
GraphicsArray (class in sage.plot.multigraphics), 423
GraphicsArray () (in module sage.plot.graphics), 422
GraphPlot (class in sage.graphs.graph_plot), 289

H

Histogram (class in sage.plot.histogram), 280
histogram() (in module sage.plot.histogram), 281
hls() (sage.plot.colors.Color method), 156
hs1() (sage.plot.colors.Color method), 157
hsv() (sage.plot.colors.Color method), 157
html_color() (sage.plot.colors.Color method), 157
html_to_float() (in module sage.plot.colors), 161
hue() (in module sage.plot.colors), 161
hyperbolic_arc() (in module sage.plot.hyperbolic_arc), 386
hyperbolic_polygon() (in module sage.plot.hyperbolic_polygon), 387
hyperbolic_regular_polygon() (in module sage.plot.hyperbolic_regular_polygon), 392
hyperbolic_triangle() (in module sage.plot.hyperbolic_polygon), 387
HyperbolicArc (class in sage.plot.hyperbolic_arc), 385
HyperbolicPolygon (class in sage.plot.hyperbolic_polygon), 387
HyperbolicRegularPolygon (class in sage.plot.hyperbolic_regular_polygon), 391

I

implicit_plot() (in module sage.plot.contour_plot), 222
inset() (sage.plot.graphics.Graphics method), 403
inset() (sage.plot.multigraphics.MultiGraphics method), 436
interactive() (sage.plot.animate.Animation method), 172
is_Graphics() (in module sage.plot.graphics), 422

L

layout_tree() (sage.graphs.graph_plot.GraphPlot method), 290
legend() (sage.plot.graphics.Graphics method), 405
lighter() (sage.plot.colors.Color method), 158
Line (class in sage.plot.line), 360
line() (in module sage.plot.line), 360
line2d() (in module sage.plot.line), 361
list_plot() (in module sage.plot.plot), 33
list_plot_loglog() (in module sage.plot.plot), 47
list_plot_semilogx() (in module sage.plot.plot), 49
list_plot_semilogy() (in module sage.plot.plot), 49
load_maps() (sage.plot.colors.Colormaps method), 158

M
make_image() (sage.plot.animate.Animation method), 172
matplotlib() (sage.plot.graphics.Graphics method), 407
matplotlib() (sage.plot.multigraphics.MultiGraphics method), 436
matrix_plot() (in module sage.plot.matrix_plot), 316
MatrixPlot (class in sage.plot.matrix_plot), 316
minmax_data() (in module sage.plot.plot), 52
mod_one() (in module sage.plot.colors), 162
module
    sage.graphs.graph_plot, 285
    sage.plot.animate, 165
    sage.plot.arc, 323
    sage.plot.arrow, 325
    sage.plot.bar_chart, 283
    sage.plot.bezier_path, 338
    sage.plot.circle, 344
    sage.plot.colors, 155
    sage.plot.complex_plot, 177
    sage.plot.contour_plot, 185
    sage.plot.density_plot, 252
    sage.plot.disk, 356
    sage.plot.ellipse, 358
    sage.plot.graphics, 397
    sage.plot.histogram, 280
    sage.plot.hyperbolic_arc, 385
    sage.plot.hyperbolic_polygon, 387
    sage.plot.hyperbolic_regular_polygon, 391
    sage.plot.line, 360
    sage.plot.matrix_plot, 316
    sage.plot.misc, 446
    sage.plot.multigraphics, 423
    sage.plot.plot, 1
    sage.plot.plot_field, 260
    sage.plot.point, 364
    sage.plot.polygon, 369
    sage.plot.primitive, 444
    sage.plot.scatter_plot, 278
    sage.plot.step, 280
    sage.plot.streamline_plot, 271
    sage.plot.text, 143
multi_graphics() (in module sage.plot.plot), 54
MultiGraphics (class in sage.plot.multigraphics), 428

N
ncols() (sage.plot.multigraphics.GraphicsArray method), 426
nrows() (sage.plot.multigraphics.GraphicsArray method), 426
options() (sage.plot.primitive.GraphicPrimitive method), 444

parametric_plot() (in module sage.plot.plot), 56
plot() (in module sage.plot.plot), 64
plot() (sage.graphs.graph_plot.GraphPlot method), 290
plot() (sage.plot.graphics.Graphics method), 407
plot() (sage.plot.multigraphics.MultiGraphics method), 440
plot3d() (sage.plot.arc.Arc method), 324
plot3d() (sage.plot.arrow.Arrow method), 326
plot3d() (sage.plot.bezier_path.BezierPath method), 339
plot3d() (sage.plot.circle.Circle method), 345
plot3d() (sage.plot.disk.Disk method), 356
plot3d() (sage.plot.ellipse.Ellipse method), 359
plot3d() (sage.plot.graphics.Graphics method), 408
plot3d() (sage.plot.line.Line method), 360
plot3d() (sage.plot.point.Point method), 364
plot3d() (sage.plot.polygon.Polygon method), 369
plot3d() (sage.plot.primitive.GraphicPrimitive method), 444
plot3d() (sage.plot.text.Text method), 143
plot_loglog() (in module sage.plot.plot), 124
plot_semilogx() (in module sage.plot.plot), 127
plot_semilogy() (in module sage.plot.plot), 131
plot_slope_field() (in module sage.plot.plot_field), 260
plot_step_function() (in module sage.plot.step), 280
plot_vector_field() (in module sage.plot.plot_field), 260
PlotField (class in sage.plot.plot_field), 260
png() (sage.plot.animate.Animation method), 173
Point (class in sage.plot.point), 364
point() (in module sage.plot.plot), 365
point2d() (in module sage.plot.plot), 366
points() (in module sage.plot.plot), 368
polar_plot() (in module sage.plot.plot), 131
Polygon (class in sage.plot.polygon), 369
polygon() (in module sage.plot.polygon), 371
polygon2d() (in module sage.plot.polygon), 372
position() (sage.plot.multigraphics.GraphicsArray method), 428
position() (sage.plot.multigraphics.MultiGraphics method), 441
rainbow() (in module sage.plot.colors), 162
reshape() (in module sage.plot.plot), 142
rgb() (sage.plot.colors.Color method), 158
rgbcolor() (in module sage.plot.colors), 163
sage.graphs.graph_plot
module, 285
sage.plot.animate
module, 165
sage.plot.arc
 module, 323
sage.plot.arrow
 module, 325
sage.plot.bar_chart
 module, 283
sage.plot.bezier_path
 module, 338
sage.plot.circle
 module, 344
sage.plot.colors
 module, 155
sage.plot.complex_plot
 module, 177
sage.plot.contour_plot
 module, 185
sage.plot.density_plot
 module, 252
sage.plot.disk
 module, 356
sage.plot.ellipse
 module, 358
sage.plot.graphics
 module, 397
sage.plot.histogram
 module, 280
sage.plot.hyperbolic_arc
 module, 385
sage.plot.hyperbolic_polygon
 module, 387
sage.plot.hyperbolic_regular_polygon
 module, 391
sage.plot.line
 module, 360
sage.plot.matrix_plot
 module, 316
sage.plot.misc
 module, 446
sage.plot.multigraphics
 module, 423
sage.plot.plot
 module, 1
sage.plot.plot_field
 module, 260
sage.plot.point
 module, 364
sage.plot.polygon
 module, 369
sage.plot.primitive
module, 444
sage.plot.scatter_plot
    module, 278
sage.plot.step
    module, 280
sage.plot.streamline_plot
    module, 271
sage.plot.text
    module, 143
save() (sage.plot.animate.Animation method), 173
save() (sage.plot.graphics.Graphics method), 408
save() (sage.plot.multigraphics.MultiGraphics method), 441
save_image() (sage.plot.graphics.Graphics method), 409
save_image() (sage.plot.multigraphics.MultiGraphics method), 441
scatter_plot() (in module sage.plot.scatter_plot), 279
ScatterPlot (class in sage.plot.scatter_plot), 278
SelectiveFormatter() (in module sage.plot.plot), 27
set_aspect_ratio() (sage.plot.graphics.Graphics method), 409
set_axes_range() (sage.plot.graphics.Graphics method), 410
set_default() (sage.plot.animate.APngAssembler method), 168
set_edges() (sage.graphs.graph_plot.GraphPlot method), 306
set_flip() (sage.plot.graphics.Graphics method), 410
set_legend_options() (sage.plot.graphics.Graphics method), 411
set_options() (sage.plot.primitive.GraphicPrimitive method), 445
set_pos() (sage.graphs.graph_plot.GraphPlot method), 311
set_vertices() (sage.graphs.graph_plot.GraphPlot method), 312
set_zorder() (sage.plot.primitive.GraphicPrimitive method), 445
setup_for_eval_on_grid() (in module sage.plot.misc), 446
show() (sage.graphs.graph_plot.GraphPlot method), 312
show() (sage.plot.animate.Animation method), 174
show() (sage.plot.graphics.Graphics method), 412
show() (sage.plot.multigraphics.MultiGraphics method), 442
streamline_plot() (in module sage.plot.streamline_plot), 271
StreamlinePlot (class in sage.plot.streamline_plot), 271

T
Text (class in sage.plot.text), 143
text() (in module sage.plot.text), 143
tick_label_color() (sage.plot.graphics.Graphics method), 421
to_float_list() (in module sage.plot.plot), 143
to_mpl_color() (in module sage.plot.colors), 164

U
unify_arguments() (in module sage.plot.misc), 448

X
xmax() (sage.plot.graphics.Graphics method), 421
xmin() (sage.plot.graphics.Graphics method), 422
xydata_from_point_list() (in module sage.plot.plot), 143
Y

ymax() (sage.plot.graphics.Graphics method), 422
ymin() (sage.plot.graphics.Graphics method), 422