Sage Vorlesung 2

last edited 05.11.2014 10:33:33 by admin

```
# Nachtrag: Löschen einer Variablen
a=0;b=0;a;b
```

auswerten

```
0
0
```

```
del(a)
```

```
b
```

```
0
```

```
a
```

```
Traceback (click to the left of this block for traceback)
...
NameError: name 'a' is not defined
```

```
# Schleifen
```

```
# for
```

```
for i in range(0,3):
    print i
```

```
0
1
2
```

```
for i in range(3,7):
    print i-1
```

```
2
3
4
5
```

```
for i in [1,1,2,4,5,1]:
    print i
```

```
1
1
2
4
5
1
```

```
for i in range(1,10,2):
    print i
```

```
1
3
5
7
9
```

```
for i in range(1,5):
    print '%4s %4s %4s' %(i,i^2,i^2)
```

```
       1    1    1
       2    4    4
       3    9    9
       4   16   16
```

```
s=0
for i in range(1,20):
    s=s+i^2
    s
```

```
    1
    5
    14
    30
    55
    91
    140
    204
    285
    385
    506
    650
    819
    1015
    1240
    1496
    1785
    2109
    2470
```

```
s
```

```
    2470
```

```
# Binomialkoeffizienten
```

```
binomial(5,2)
```

```
    10
```

```
5*4*3*2*1/((1*2)*(1*2*3))
```

```
    10
```

```
factorial(5)/factorial(2)/factorial(5-2)
```

```
    10
```

```
# Pascalsches Dreieck
# In n-ter Zeile steht
# binomial(n,0) , ..., binomial(n,n)
for i in range(0,6):
    L=[binomial(i,j) for j in range(0,i+1)]
    print(L)
```

```
    [1]
    [1, 1]
    [1, 2, 1]
    [1, 3, 3, 1]
    [1, 4, 6, 4, 1]
    [1, 5, 10, 10, 5, 1]
```

```
summe=sum(L); summe
```

```
    32
```

```
factor(summe)
```

```
    2^5
```

```
n,i=var('n','i')
sum(binomial(n,i),i,0,n)
```

```
    2^n
```

```
# while
```

```
i=0
while i<5:
    print(i)
    i=i+1
```

```
    0
    1
    2
    3
    4
```

```
s=0;i=0
while i<5:
    s=s+i^2
    i=i+1
    s
```

```
    0
    1
    5
    14
    30
```

```
# Verzweigungen
```

```
a=1
if (a>=0):
    print(n(sin(a)))
else:
    print(a)
```

```
    0.841470984807897
```

```
s=0
for i in range(0,10):
    if (i<5):
        s=1
    else:
        s=s+1
s
```

```
    6
```

```
b=-1.0
if (b>0):
    print(`b`+' ist positiv')
elif (b==0):
    print('%f ist Null')%b
else:
    print('%i ist negativ')%b
```

```
    -1 ist negativ
```

```
# Funktionen definieren
```

```
def quadrat(x):
    print(x)
    return x^2
```

```
quadrat(2)
```

```
    2
    4
```

```
a
```

```
    4
```

```
def potenzen(x):
    a=x^2
    b=x^3
    return a,b
```

```
[y,z]=potenzen(x);[yy,zz]=potenzen(2)
```

```
y;z;yy;zz
```

```
    x^2
    x^3
    4
    8
```

```
is_even(2)
```

```
    True
```

```
def my_is_even(n):
    return n%2==0
```

```
my_is_even(2); my_is_even(3)
```

```
    True
    False
```

```
# Umrechnung Fahrenheit nach Celsius
```

```
def C(x):
    return 5/9*x-160/9
```

```
C(50)
```

```
    10
```

```
for i in range(30,100,10):
    print n(C(i))
```

```
    -1.11111111111111
    4.44444444444444
    10.0000000000000
    15.5555555555556
    21.1111111111111
    26.6666666666667
    32.2222222222222
```

```
reset()
```

```
# Ausgabe aller geraden Zahlen
```

```
def my_even_numbers(y):
    if (y in ZZ): # ZZ ist Menge der ganzen Zahlen
        L=[]
        for i in range(1,y+1):
            if my_is_even(i):
                L=L+[i]
        return L
    else:
        print('nur Eingabe ganzer Zahlen möglich')
    return L
```

```
my_even_numbers(22)
```

```
    [2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22]
```

```
def finde_quadratzahlen(L):
    Liste=[]
    for i in range(0,len(L)):
        if sqrt(L[i]) in ZZ:
            Liste=Liste+[L[i]]
    return Liste
```

```
finde_quadratzahlen(my_even_numbers(101))
```

```
    [4, 16, 36, 64, 100]
```

```
def monte_carlo(n):
    print ('Monte-Carlo Methode zur Näherung von pi')
    v=0
    for i in range(1,n+1):
        x=random() # (0,1)-gleichverteilte Zufallszahl
        y=random()
        if (x^2+y^2 <=1):
            v=v+1
    pi_schaetzer=4*(v/n).n()
    print ('Die Näherung beträgt %f')%pi_schaetzer
```

```
monte_carlo(10000)
```

```
    Monte-Carlo Methode zur Näherung von pi
    Die Näherung beträgt 3.138000
```

```
# Debuggen
```

```
def finde_quadratzahlen(L):
    Liste=[]
    for i in range(0,len(L)):
        if sqrt(L[i]) in ZZ:
            Liste=Liste+[L[i]]
        else:
            a=1/0
    return Liste
```

```
finde_quadratzahlen(10)
```

```
    Traceback (click to the left of this block for traceback)
    ...
    TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

```
finde_quadratzahlen([2,4,6])
```

```
    Traceback (click to the left of this block for traceback)
    ...
    ZeroDivisionError: Rational division by zero
```

```
def finde_quadratzahlen(L):
    Liste=[]
    for i in range(0,len(L))
        if sqrt(L[i]) in ZZ:
            Liste=Liste+[L[i]]
    return Liste
```

```
    Traceback (click to the left of this block for traceback)
    ...
    SyntaxError: invalid syntax
```

```
def finde_quadratzahlen(L):
    Liste=[]
    for i in range(0,len(L)):
    if sqrt(L[i]) in ZZ:
            Liste=Liste+[L[i]]
    return Liste
```

```
    Traceback (click to the left of this block for traceback)
    ...
    IndentationError: expected an indented block
```

```
# Wenn Funktion kompiliert, aber Ergebnis fehlerhaft / Fehlermeldung, dann über print Zwischenergebnisse
```

```
# rekursive Funktionen
```

```
factorial(5)
```

```
    120
```

```
def fact(y):
    fact=1
    for i in range(1,y+1):
        fact=i*fact
    return fact
```

```
fact(5)
```

```
    120
```

```
def fact(y):
    if y==1:
        return 1
    else:
        return fact(y-1)*y
```

```
fact(5)
```

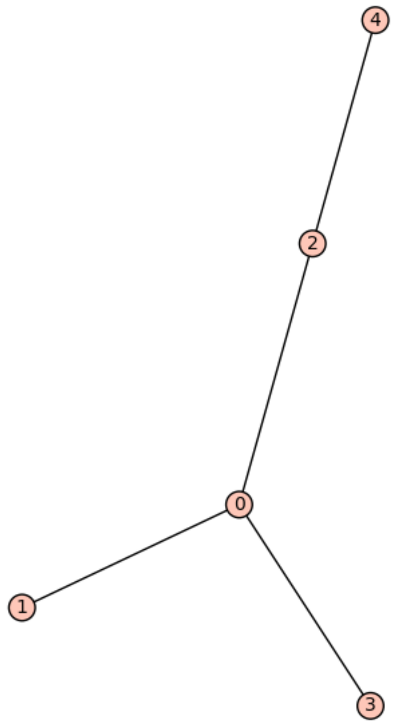
```
    120
```

```
# Graphentheorie
```

```
g = Graph({0:[1,2,3], 2:[4]}); g
```

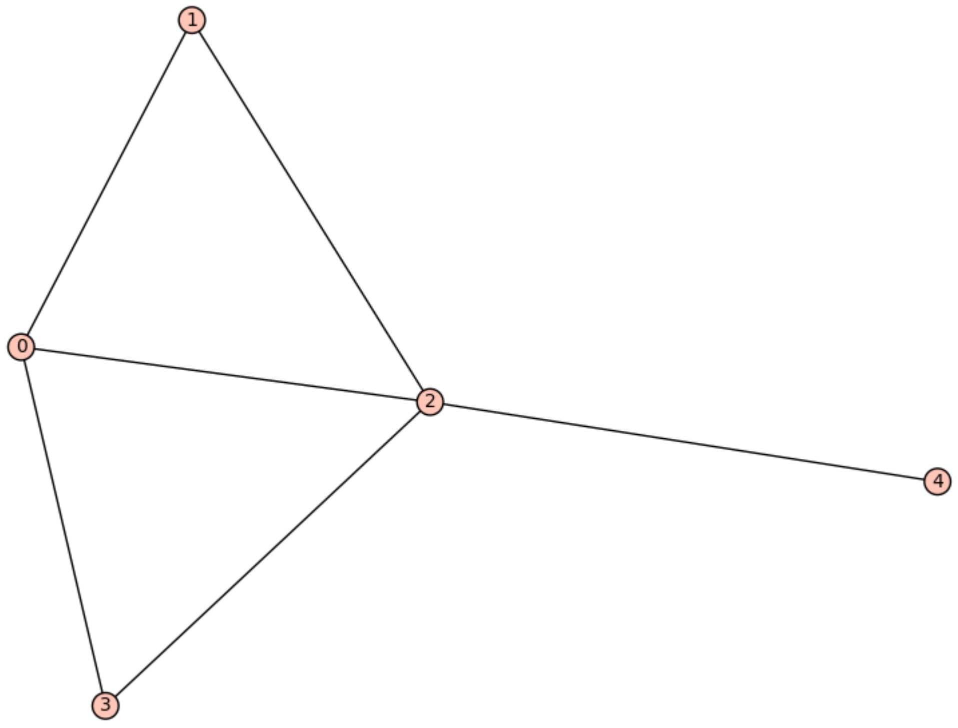
```
    Graph on 5 vertices
```

```
g.plot()
```



```
g.add_edge(1,2)
g.add_edge(2,3)
```

```
g.plot()
```

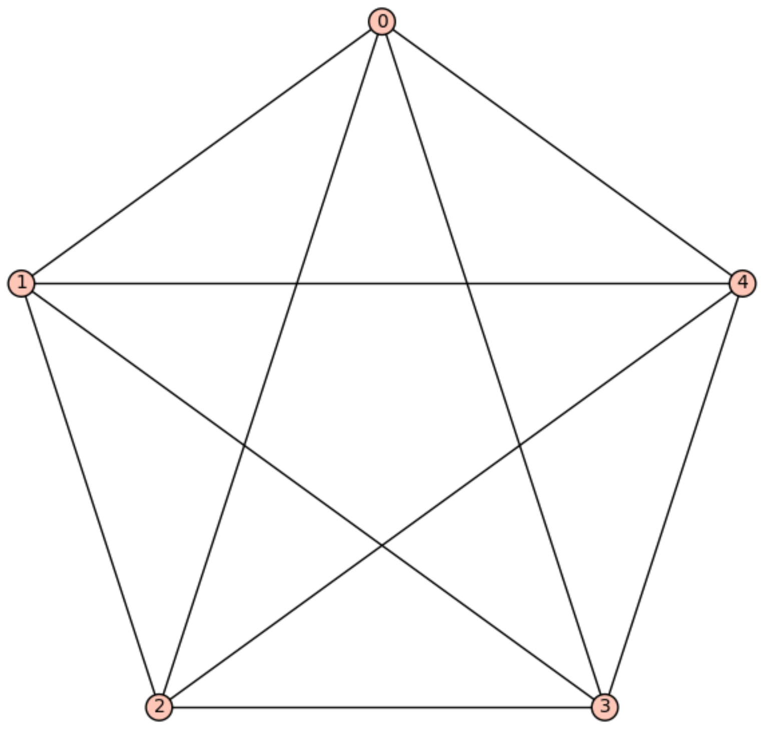


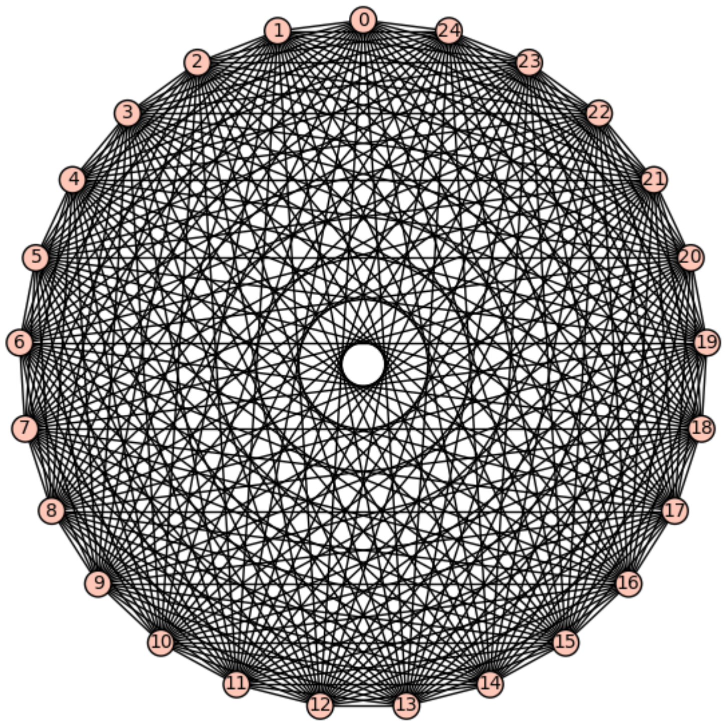
```
g.is_planar()
```

```
True
```

```
g=graphs.CompleteGraph(5);h=graphs.CompleteGraph(25)
```

```
g.plot();h.plot()
```

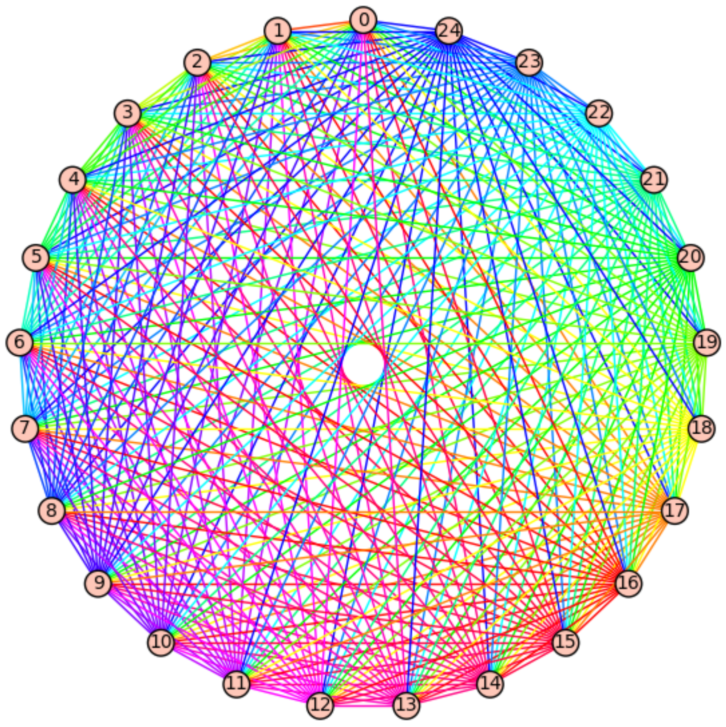




```
# Funktion erstellen, die einen Graph erzeugt
def mygraph(n):
    G=Graph({})
    for i in range(0,n):
        for j in range(0,n):
            if (i<j):
                G.add_edge(i,j,j) # dritter Parameter: Bezeichnung
    return(G)
```

```
G=mygraph(25)
```

```
G.plot(color_by_label=True, layout='circular')
```



```
# Funktionen
```

```
def f(x):
    if x>=0:
        return(sin(x))
    else:
        return(x)
```

```
f(-1);f(pi)
```

```
-1
0
```

```
g=x^2; g
```

```
x^2
```

```
g(1)
```

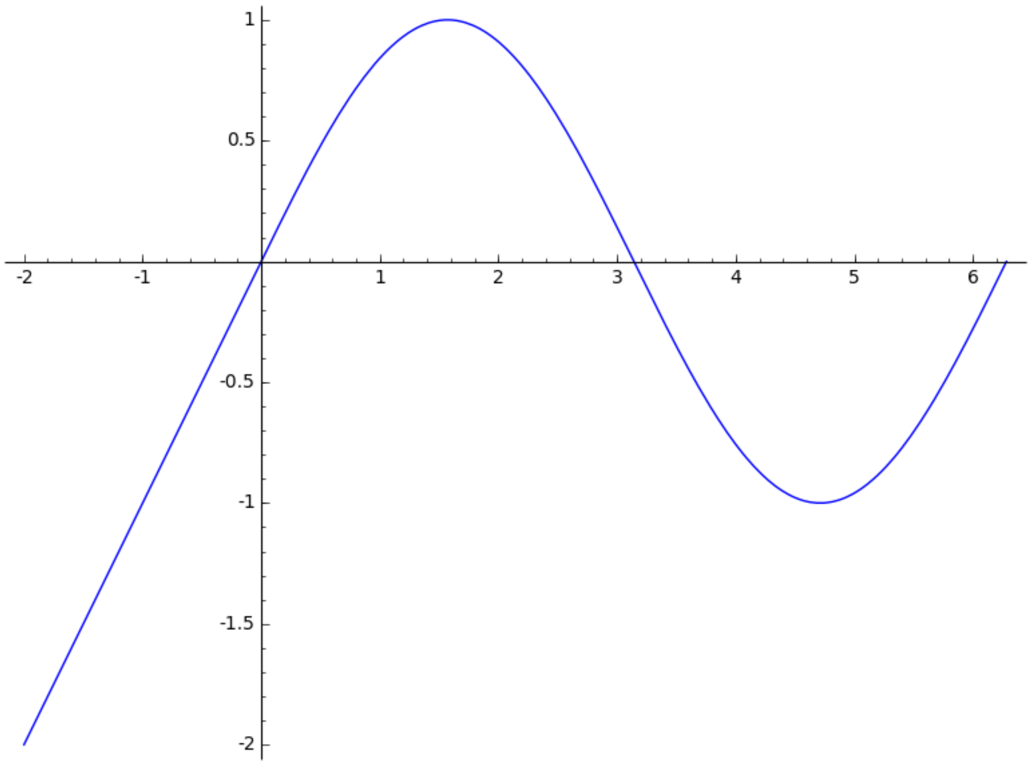
```
1
```

```
g(-1)
```

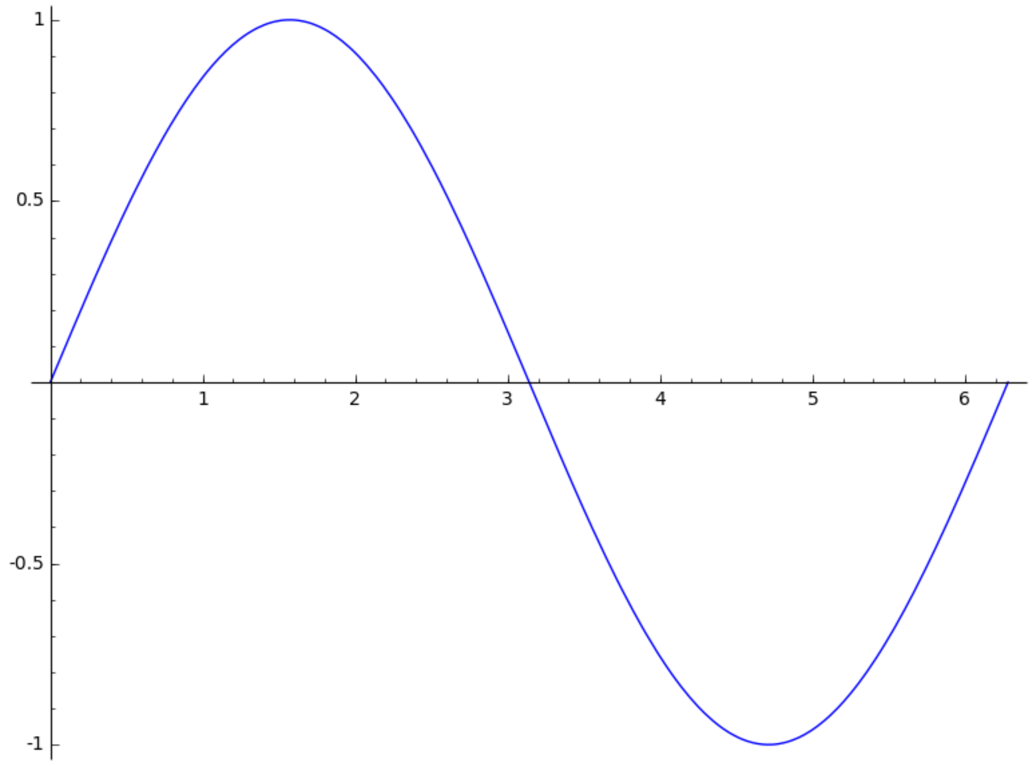
```
1
```

```
# Plotten
```

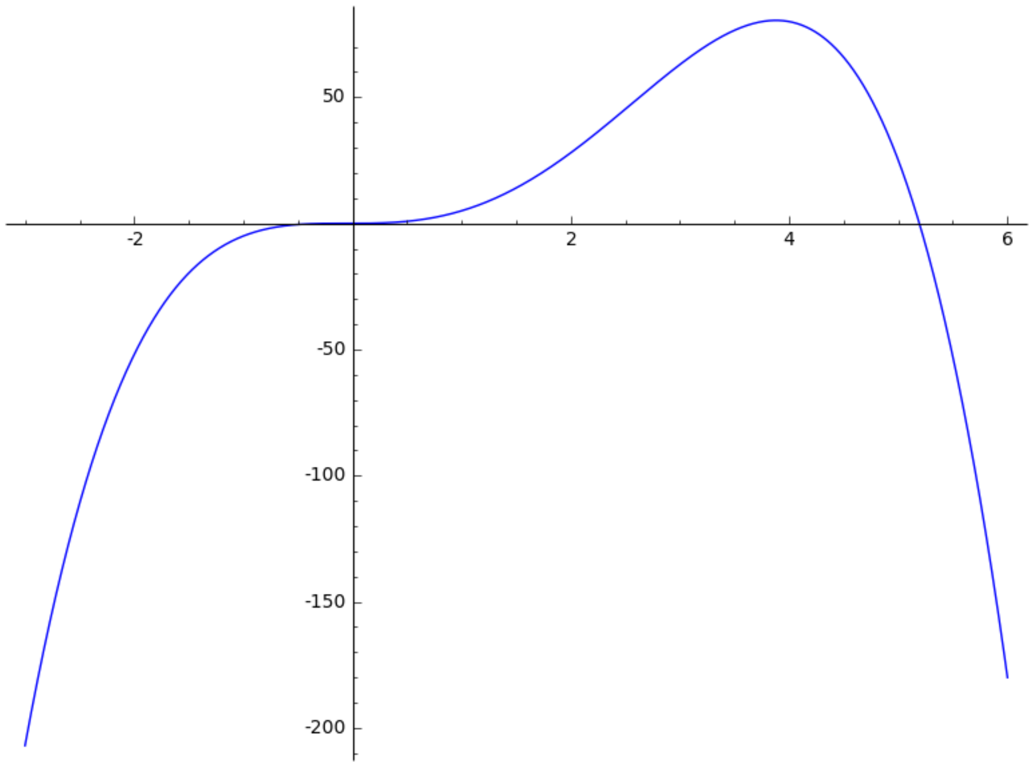
```
plot(f,(-2,2*pi))
```



```
plot(sin,(0,2*pi))
```



```
plot(g+5*x^3-x^4,(-3,6))
```



```
h=g+5*x^3-x^4
```

```
h.find_local_maximum(0,6);
```

```
(80.4748381080422, 3.8789024348044085)
```

```
plot?
```

File: /home/sage/sage-6.3/local/lib/python2.7/site-packages/sage/misc/decorators.py

Type: <type 'function'>

Definition: plot(funcs, exclude=None, fillalpha=0.5, fillcolor='automatic', detect_poles=False, plot_points=200, thickness=1, ada
adaptive_recursion=5, aspect_ratio='automatic', alpha=1, legend_label=None, fill=False, *args, **kwds)

Docstring:

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
`GraphicPrimitive` object.

There are many other specialized 2D plot commands available in Sage, such as `plot_slope_field`, as well as various graph
`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
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 refineme
refinement.
- `adaptive_tolerance` - (default: 0.01) how large a difference should be before the adaptive refinement code considers
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
`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"`, `"se`

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

Note

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

APPEARANCE OPTIONS:

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

INPUT:

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

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

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

  The linestyle can also be prefixed with a drawing style (e.g., `"steps--"`)

    - `"default"` (connect the points with straight lines)
    - `"steps"` or `"steps-pre"` (step function; horizontal line is to the left of point)
    - `"steps-mid"` (step function; points are in the middle of horizontal lines)
    - `"steps-post"` (step function; horizontal line is to the right of point)

- `marker` - The style of the markers, which is one of
    - `"None"` or `" "` or `""` (nothing) – default
    - `","` (pixel), `"."` (point)
    - `"_"` (horizontal line), `"|"` (vertical line)
    - `"o"` (circle), `"p"` (pentagon), `"s"` (square), `"x"` (x), `"+"` (plus), `"*"` (star)
    - `"D"` (diamond), `"d"` (thin diamond)
    - `"H"` (hexagon), `"h"` (alternative hexagon)
    - `"<"` (triangle left), `">"` (triangle right), `"^"` (triangle up), `"v"` (triangle down)
    - `"1"` (tri down), `"2"` (tri up), `"3"` (tri left), `"4"` (tri right)
    - `0` (tick left), `1` (tick right), `2` (tick up), `3` (tick down)
    - `4` (caret left), `5` (caret right), `6` (caret up), `7` (caret down)
    - `"$...$"` (math TeX string)

- `markersize` - the size of the marker in points

- `markeredgecolor` – the color of the marker edge

- `markerfacecolor` – the color of the marker face

- `markeredgewidth` - the size of the marker edge in points

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

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`
      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`
      function in the list and the horizontal line y = j.)
- `fillcolor` - (default: ‘automatic’) The color of the fill. Either ‘automatic’ or a color.
- `fillalpha` - (default: 0.5) How transparent the fill is. A number between 0 and 1.

Note

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

EXAMPLES:

We plot the $\sin$ function:

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

```
225
sage: P          # render
```

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

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

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

Some colored functions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
sage: G = plot(exp, 1, 10) # long time
```

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

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

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

You have the function (in blue) and its approximation (in green) passing through the points A and B. The algorithm finds the r
distance between C and D. If that distance exceeds the `adaptive_tolerance` threshold (*relative* to the size of the initial plot
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 repeat
by default).

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

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

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

We can also directly plot the elliptic curve:

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

We can change the line style as well:

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

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

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

The marker can be a TeX symbol as well:

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

Sage currently ignores points that cannot be evaluated

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

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

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

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

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

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

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

We can detect the poles of a function:

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

We draw the Gamma-Function with its poles highlighted:

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

The basic options for filling a plot:

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

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

A example about the growth of prime numbers:

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

Fill the area between a function and its asymptote:

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

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

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

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

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

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

```
sage: plot(sin(x^2), (x, -3, 3), title='Plot of $\sin(x^2)$', axes_labels=['$x$','$y$']) # These
sage: plot(sin(x^2), (x, -3, 3), title='Plot of sin(x^2)', axes_labels=['x','y']) # These will no
```

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

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

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

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

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

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

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

You can force Type 1 fonts in your figures by providing the relevant option as shown below. This also requires that LaTeX, dv

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

A example with excluded values:

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

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

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

Excluded points can also be given by an equation:

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

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

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

Regions in which the plot has no values are automatically excluded. The regions thus excluded are in addition to the exclusior
keyword argument.:

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

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

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

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

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

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

TESTS:

We do not randomize the endpoints:

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

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

```
sage: d = plot([sin(x), cos(x)], 100, 120).get_minmax_data()
sage: d['xmin']
100.0
sage: d['xmax']
120.0
```

We check various combinations of tuples and functions, ending with tests that lambda functions work properly with explicit v

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

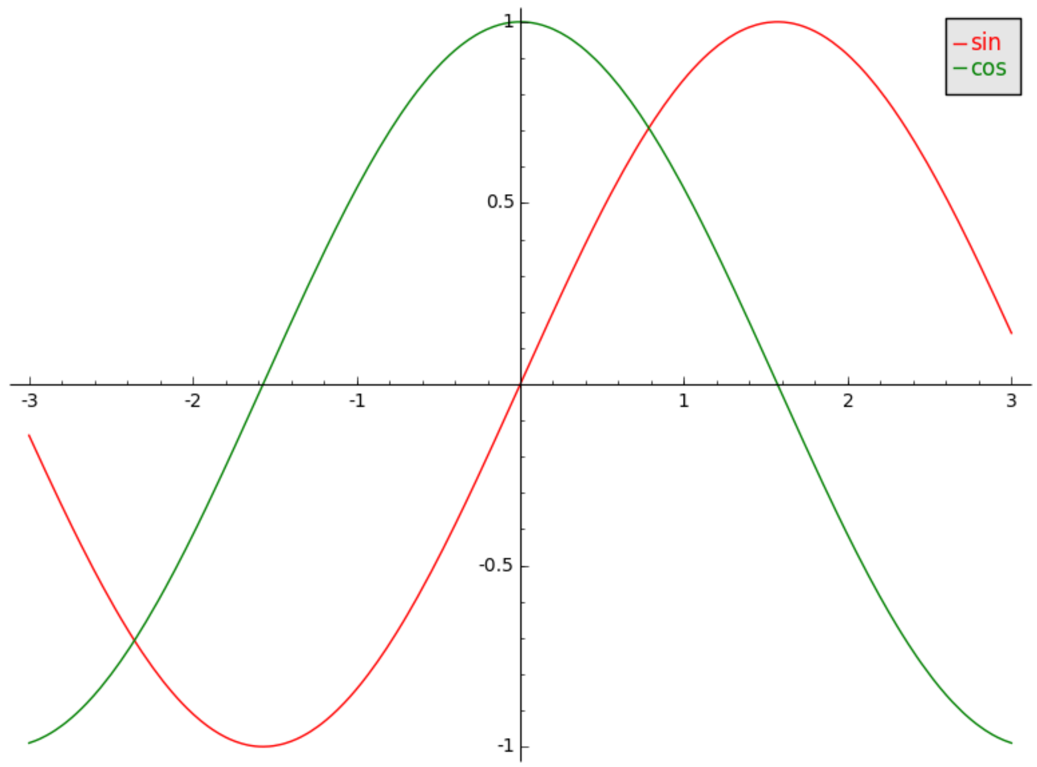
We check to handle cases where the function gets evaluated at a point which causes an ‘inf’ or ‘-inf’ result to be produced.

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

Bad options now give better errors:

```
sage: P = plot(sin(1/x), (x,-1,3), foo=10)
Traceback (click to the left of this block for traceback)
...
```

```
sage: p1 = plot(sin(x), (x,-3,3),legend_label='sin', legend_color='red', color='red')
sage: p2 = plot(cos(x), (x,-3,3),legend_label='cos', legend_color='green', color='green')
sage: p1 + p2
```



```
# Flächen plotten
```

```
F1=[[1,0],[2,0],[2,1],[1,1]]
```

```
disp?
```

File: /home/sage/sage-6.3/local/lib/python2.7/site-packages/sage/plot/graphics.py

Type: <class ‘sage.plot.graphics.Graphics’>

Definition: disp([noargspec])

Docstring:

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

EXAMPLES:

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

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

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

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

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

TESTS:

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

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

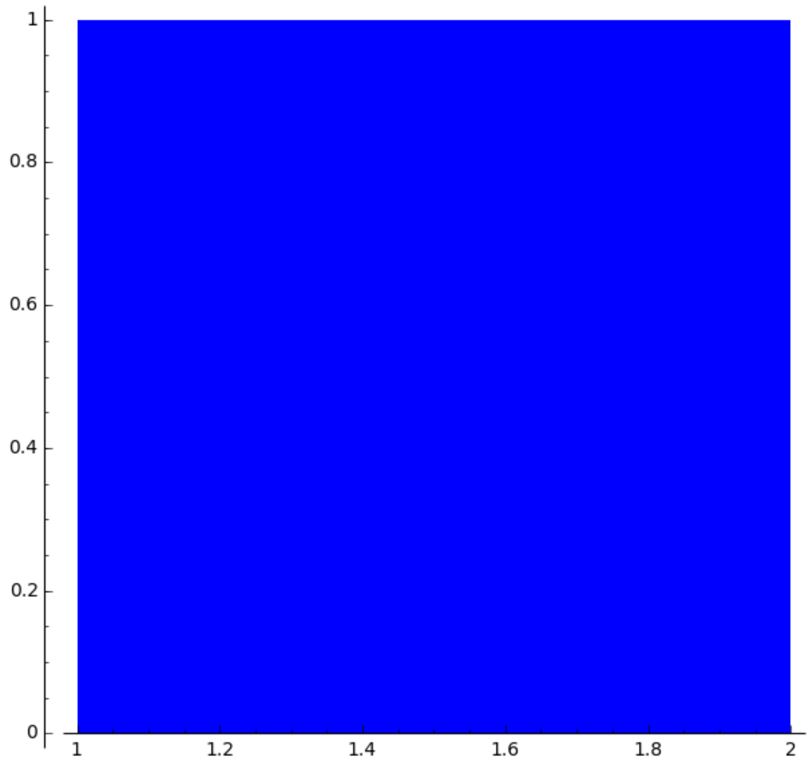
We check that graphics can be pickled (we can’t use equality on graphics so we just check that the load/dump cycle gives a `Graphics` instance):

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

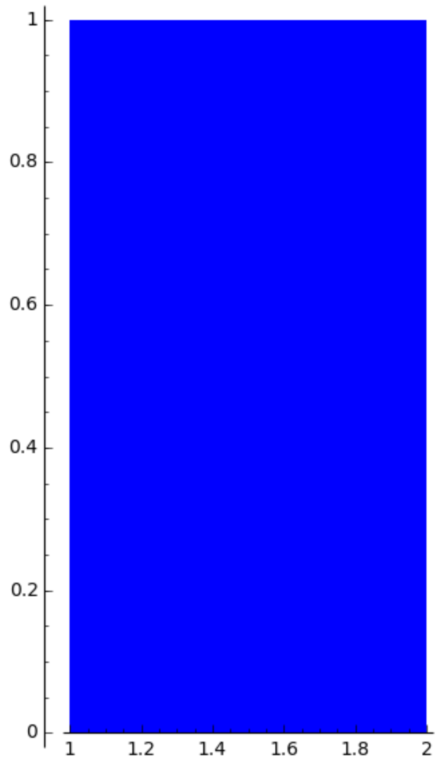
```
sage: isinstance(g2, Graphics)
True
```

```
disp=polygon2d(F1)
```

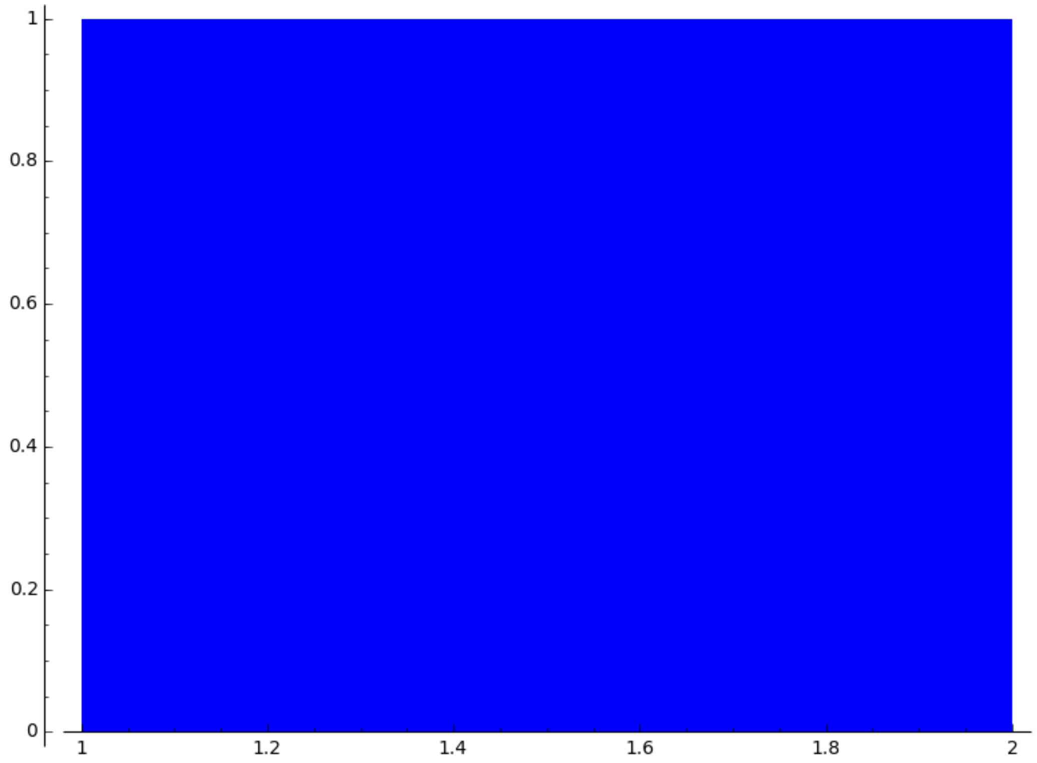
```
disp
```



```
disp.show(aspect_ratio=2)
```

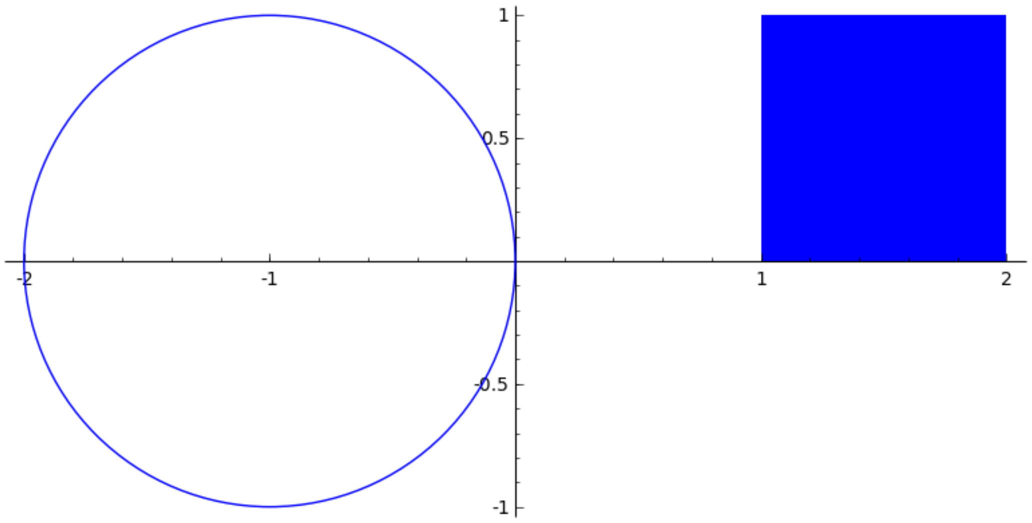


```
plot(disp)
```



```
# Kreis hinzufügen
```

```
disp=polygon2d(F1) + circle((-1,0),1)
disp.show(aspect_ratio=1)
```



```
# Ableitungen
```

```
reset()
```

```
x=var('x')
```

```
diff(sin(x^2),x)
```

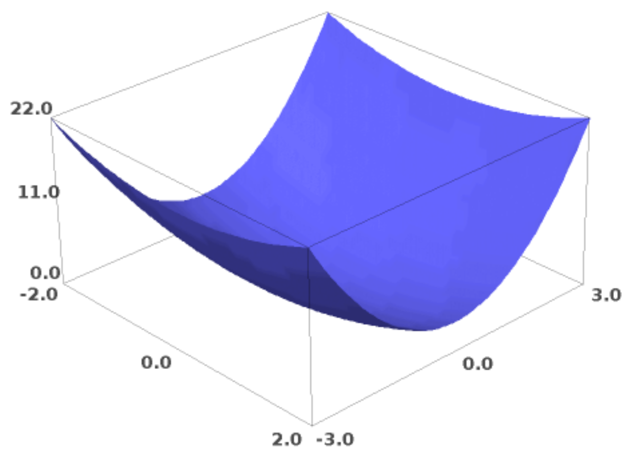
```
    2*x*cos(x^2)
```

```
y=var('y')
f=x^2+2*y^2
```

```
plot3d(f,(-2,2),(-3,3))
```

```
    Sleeping...
```

Make Interactive



```
f.diff(x)
```

```
2*x
```

```
diff(f,y,1)
```

```
4*y
```

```
f.diff(y,2) # zweite Ableitung
```

```
4
```

```
diff(sin(x^2),x,4)
```

```
16*x^4*sin(x^2) - 48*x^2*cos(x^2) - 12*sin(x^2)
```

```
f=x*sin(x^2)
```

```
q=integral(f,x,0,pi/2)
```

```
q.n()
```

```
0.890605946055244
```

```
integral(f,x)
```

```
-1/2*cos(x^2)
```