lektion4

November 5, 2025

Table of Contents

1 Lektion 4

1.1 For Schleife und List Comprehension

Der Aufbau einer for Schleife ist wie folgt

```
for element in iterierbarers_objekt:
    block von anweisungen
```

Mit `break` kann man eine Schleife vorzeitig verlassen und mit `continue` springt man zur nächsten Iterierten und überspringt den Rest.

[1]:
```
for n in range(1, 10):
    if n % 4 == 0:
        print(f'{n} teilbar durch 4: {n} = 4*{n//4}')
```

```
        continue
    if n % 2 == 0:
        print(f'{n} teilbar durch 2: {n} = 2*{n//2}')
```

```
2 teilbar durch 2: 2 = 2*1
4 teilbar durch 4: 4 = 4*1
6 teilbar durch 2: 6 = 2*3
8 teilbar durch 4: 8 = 4*2
```

[2]:
```
L = []
for i in range(6):
    L.append(i**2)
L
```

[2]: [0, 1, 4, 9, 16, 25]

[3]:
```
L = [i**2 for i in range(6)]  # Liste der Quadratzahlen
L
```

[3]: [0, 1, 4, 9, 16, 25]

[4]:
```
L = [i**2 for i in range(6) if i%2 == 1] # Liste der ungeraden Quadratzahlen
L
```

[4]: [1, 9, 25]

[5]:
```
L = [i**2 if i%2==1 else -i for i in range(6)]
L
```

[5]: [0, 1, -2, 9, -4, 25]

1.2 Kopieren von Listen

[6]:
```
a = [1, 2]
b = [3, 4, a]
b
```

[6]: [3, 4, [1, 2]]

[7]:
```
a[0] = 11
```

[8]:
```
b
```

[8]: [3, 4, [11, 2]]

[9]:
```
bb = [3, 4, a.copy()]
bb
```

[9]: [3, 4, [11, 2]]

[10]:
```
a[0] = 22
```

[11]:
```
bb
```

[11]: [3, 4, [11, 2]]

[12]:
```
b
```

[12]: [3, 4, [22, 2]]

1.3 Python Funktionen

[13]:
```
pf = lambda x : x**2
pf(2)
```

[13]: 4

das gleiche ausführlicher

[14]:
```
def mysqr2(x):
    """ berechnet x Quadrat"""
    y = x**2
    return y
```

[15]:
```
mysqr2(2)
```

[15]: 4

[16]:
```
?mysqr2
```

Mit Funktionen lassen sich Teile eines Programms zusammenfassen. Dadurch wird der Programmcode übersichtlicher und weniger anfällig für Fehler.

Ein wesentliches Merkmal von Funkttionen ist die Rückgabe von Werten an das die Funktion aufrufende Programm. Dazu kann man mit der return Anweisung Rückgaben definieren. Erreicht man innerhalb einer Funktion eine return Anweisung, so wird die Funktion verlassen und das Objekt zurückgeliefert, das nach der return Anweisung erzeugt wird. Funktionen ohne return Anweisungen geben None zurück. Funktionen haben die folgende allgemeine Struktur:

```
def funktionsname(parameter):
    anweisungen
```

Funktionen beginnen in Python mit dem Schlüsselwort def gefolgt vom Funktionsnamen. In einer Klammer folgen ein oder mehrere Parameter. Nach der schließenden Klammer steht ein Doppelpunkt. Eingerückt stehen in den weiteren Zeilen Anweisungen.

[17]:
```
def mypow(x, n=2):
    """ Berechnet x**n und falls n nicht gegeben ist das Quadrat von x"""
    y = x**n
    return y
```

[18]:
```
mypow(3)
```

[18]: 9

[19]:
```
mypow(3, 3)
```

[19]: 27

Achtung

[20]:
```
def f(a, L=[]):
    L.append(a)
    return L


print(f(1))
print(f(2))
print(f(3))
print(f(4, [2, 3, 1]))
```

```
[1]
[1, 2]
[1, 2, 3]
[2, 3, 1, 4]
```

Der default wird nur einmal ausgewertet und [] ist ein veränderbares Objekt.

Eine Lösung ist:

[21]:
```
def f(a, L=None):
    if L is None:
        L = []
    L.append(a)
    return L


print(f(1))
print(f(2))
print(f(4, [2, 3, 1]))
```

```
[1]
[2]
[2, 3, 1, 4]
```

Ein Funktion kann auf Variablen zugreifen, die auf 'höherer' Ebene definiert wurden. Und im Fall von Listen diese auch verändern.

[22]:
```
b = [7, 2]


def f(a):
    c = b[0] + a
    b[1] = 1
    return c


print(b, f(2))
```

```
[7, 1] 9
```

[23]:
```
b = 7


def f(a):

    c = b + a
    return c


print(b, f(2))
```

```
7 9
```

[24]:
```
b = 7


def f(a):
    b = 21
    c = b + a
    return c


print(b, f(2))
```

```
7 23
```

Hier ist b zwei verschiedene Variablen. Einmal eine Variable, die auf die Zahl 7 verweist und andererseits eine lokale Variable innerhalb der Funktion f, die den Wert 21 hat, die auf die Variable b, die eine Ebene höher existiert, keine Auswirkung hat.

1.4 Sympy Funktionen

[25]:
```
from sympy import *

init_printing()
x, y, z = symbols('x y z')
q = Lambda(x, x**2)  # vgl. lambda x : expr
q
```

[25]: $(x \mapsto x^2)$

[26]:
```
q(23)
```

[26]: 529

[27]:
```
f = Lambda((x, y, z), x * y + y - 2 * z**2)
f
```

[27]: $((x,\ y,\ z) \mapsto xy + y - 2z^2)$

[28]:
```
f(1, 2, 3)
```

[28]: -14

[29]:
```
param = x, y, z
type(param)
```

[29]:
```
tuple
```

[30]:
```
f(*param)  # * ist hier der "argument unpacking operator"
```

[30]: $xy + y - 2z^2$

[31]:
```
f = Function('f')
g = Function('g')
```

[32]:
```
f(g(x))
```

[32]: $f(g(x))$

[33]:
```
k = f(x) + g(1 / x)
k
```

[33]: $f(x) + g\left(\frac{1}{x}\right)$

[34]:
```
k.diff(x)
```

[34]: $\frac{d}{dx} f(x) - \frac{\frac{d}{d\xi_1} g(\xi_1)\Big|_{\xi_1=\frac{1}{x}}}{x^2}$

1.5 Numpy Funktionen

[35]:
```
import numpy as np
```

[36]:
```
xn = np.linspace(0 , 1, 4)
xn
```

[36]:
```
array([0.        , 0.33333333, 0.66666667, 1.        ])
```

[37]:
```
np.sin(xn)
```

[37]:
```
array([0.        , 0.3271947 , 0.6183698 , 0.84147098])
```

[38]:
```
np.sin(np.pi*xn)
```

[38]:
```
array([0.00000000e+00, 8.66025404e-01, 8.66025404e-01, 1.22464680e-16])
```

1.6 Lamdifizierung (sympy -> numpy/scipy)

[39]:
```
f = x**2 * sin(x)
f
```

[39]:
$$x^2 \sin(x)$$

[40]:
```
fn = lambdify(x, f)
```

[41]:
```
xn = np.linspace(0, 11, 5)
fn(xn)
```

[41]:
```
array([   0.        ,    2.88631125,  -21.34259485,   62.79474906,
       -120.99881499])
```

[42]:
```
f = Integral(exp(-x**2), (x, 1, y))
F = f.doit()
F  # erf: gaußsche Fehlerfunktion (engl. error function)
```

[42]:
$$\frac{\sqrt{\pi}\,\mathrm{erf}(y)}{2} - \frac{\sqrt{\pi}\,\mathrm{erf}(1)}{2}$$

[43]:
```
Fn = lambdify(y, F)
Fn(2)
```

[43]:
$$0.135257257949995$$

[44]:
```
%%timeit
Fn(2)
```

```
5.34 s ± 686 ns per loop (mean ± std. dev. of 7 runs, 100,000 loops each)
```

[45]:
```
%%timeit
F.subs(y, 2).evalf()
```

```
355  s ± 44  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[46]:
```
%%timeit
N(F.subs(y, 2))
```

```
291  s ± 2.72  s per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[47]:
```
Fn(xn)
```

[47]:
```
array([-0.74682413,  0.13931362,  0.13940279,  0.13940279,  0.13940279])
```

1.7 Sympy Graphik 2D

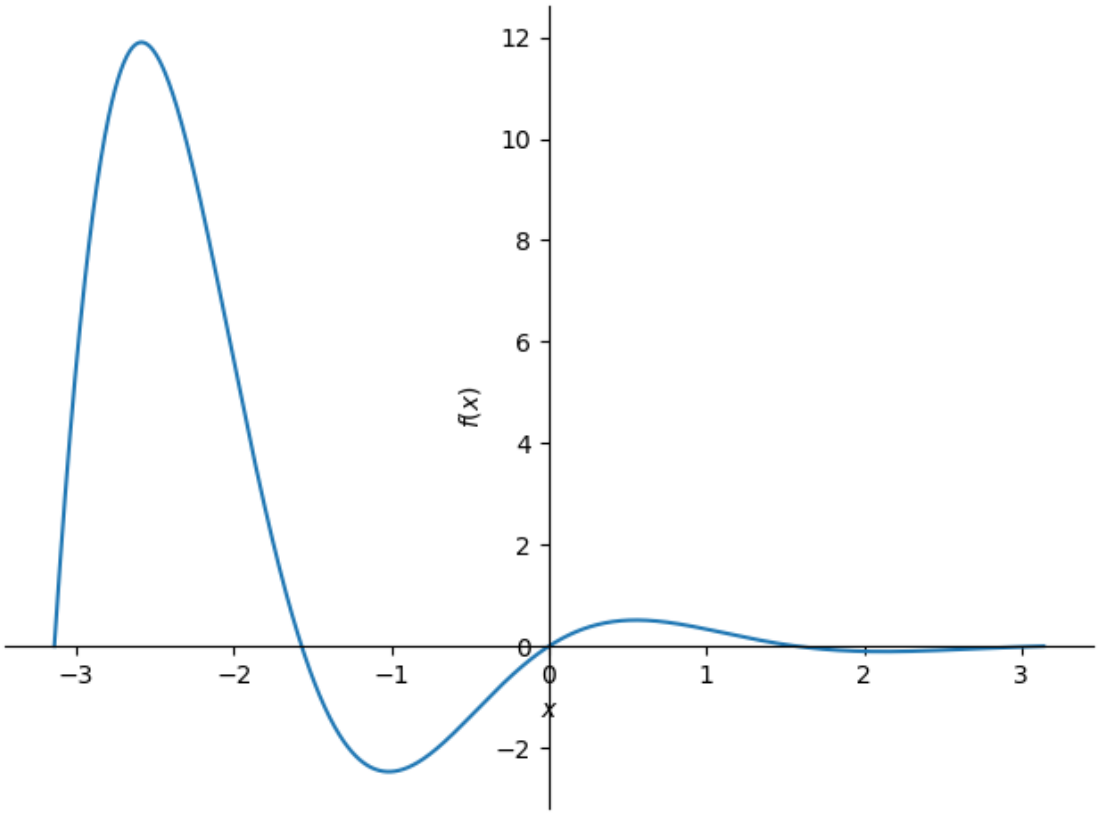
[48]:
```
import sympy as sp
import numpy as np
```

1.7.1 Einfache Graphen von Funktionen

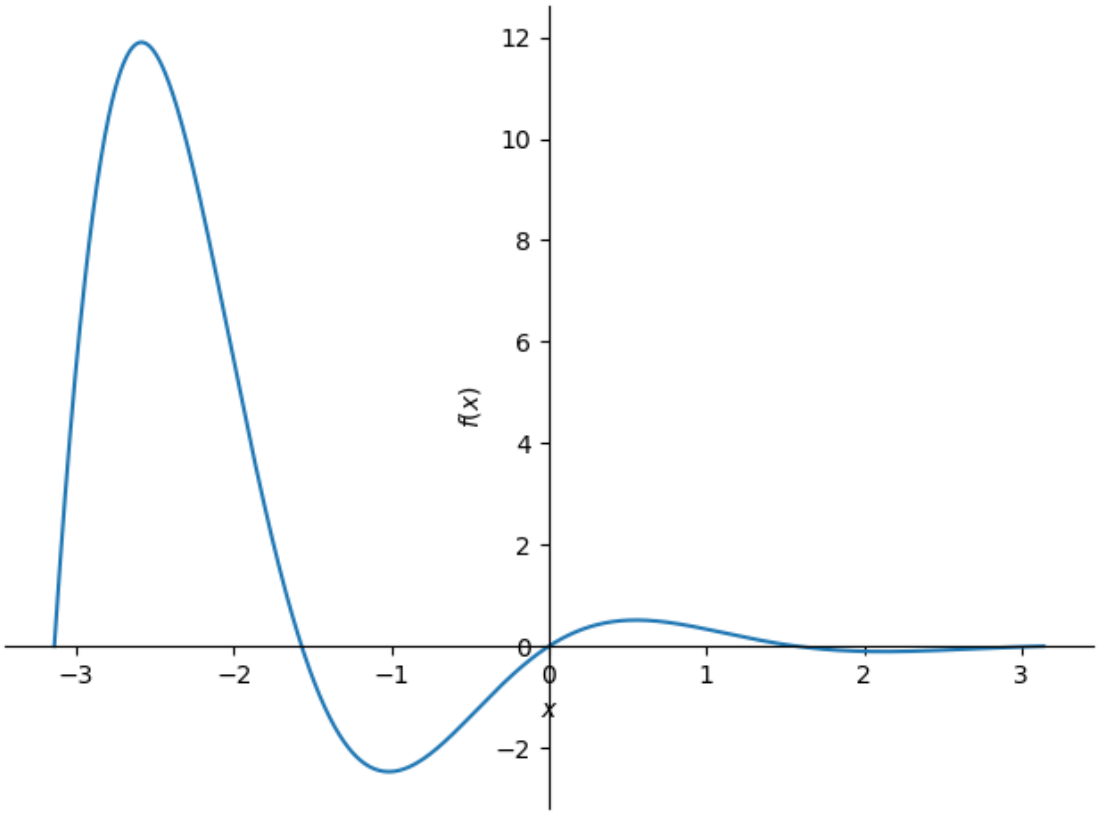
Graph: Alle Punkte (x, y) mit

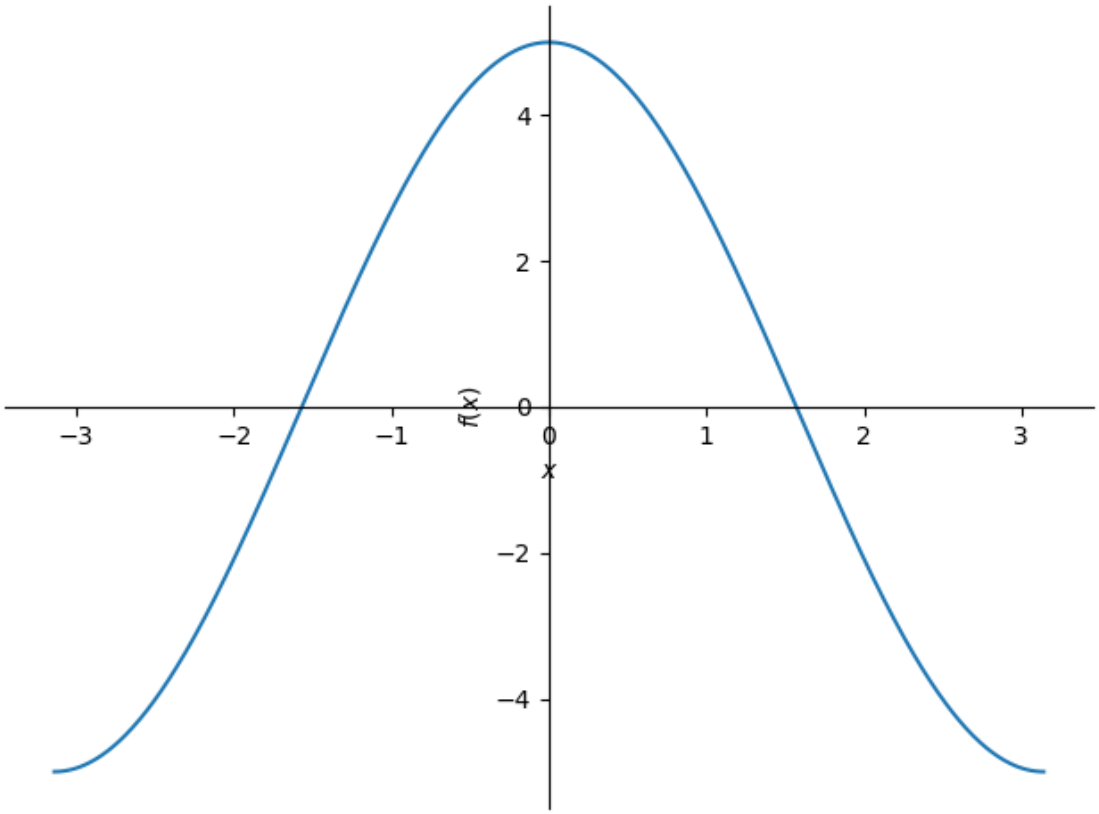
$$y = f(x)$$

[49]:
```
x = sp.symbols('x')
f = sp.exp(-x) * sp.sin(2 * x)
p1 = sp.plot(f, (x, -sp.pi, sp.pi))
```

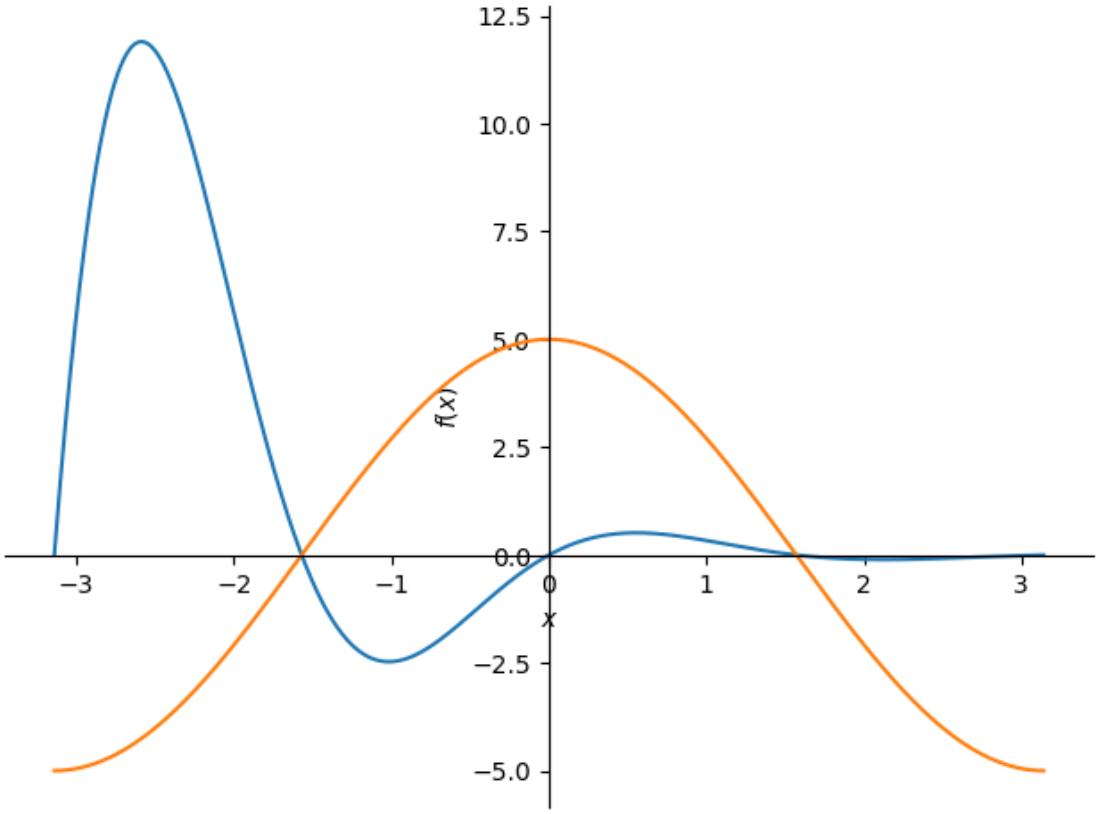


[50]:
```
p1 = sp.plot(f, (x, -sp.pi, sp.pi))
p2 = sp.plot(5 * sp.cos(x), (x, -sp.pi, sp.pi))
```

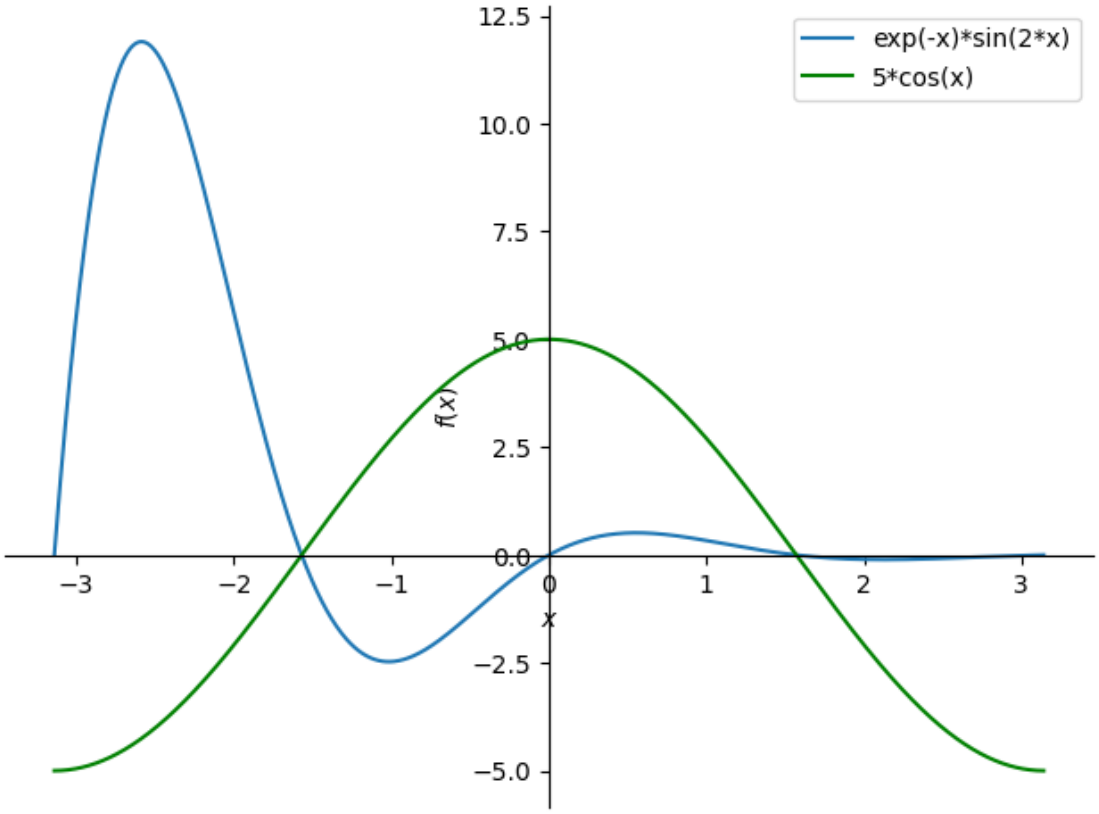

12
10
8
6
4
2
0
−2
f(x)
−3
−2
−1
0
1
2
3
x



[51]:
```
p1 = sp.plot(f, (x, -sp.pi, sp.pi), show=False)
p2 = sp.plot(5 * sp.cos(x), (x, -sp.pi, sp.pi), show=False)
p1.extend(p2)
p1.show()
```



[52]:
```
p1 = sp.plot(f, (x, -sp.pi, sp.pi), show=False)
p2 = sp.plot(5 * sp.cos(x), (x, -sp.pi, sp.pi), show=False)
p1.extend(p2)
p1.legend = True
p1[1].line_color = 'green'
p1.show()
```



1.7.2 Implizit gegebene Kurven

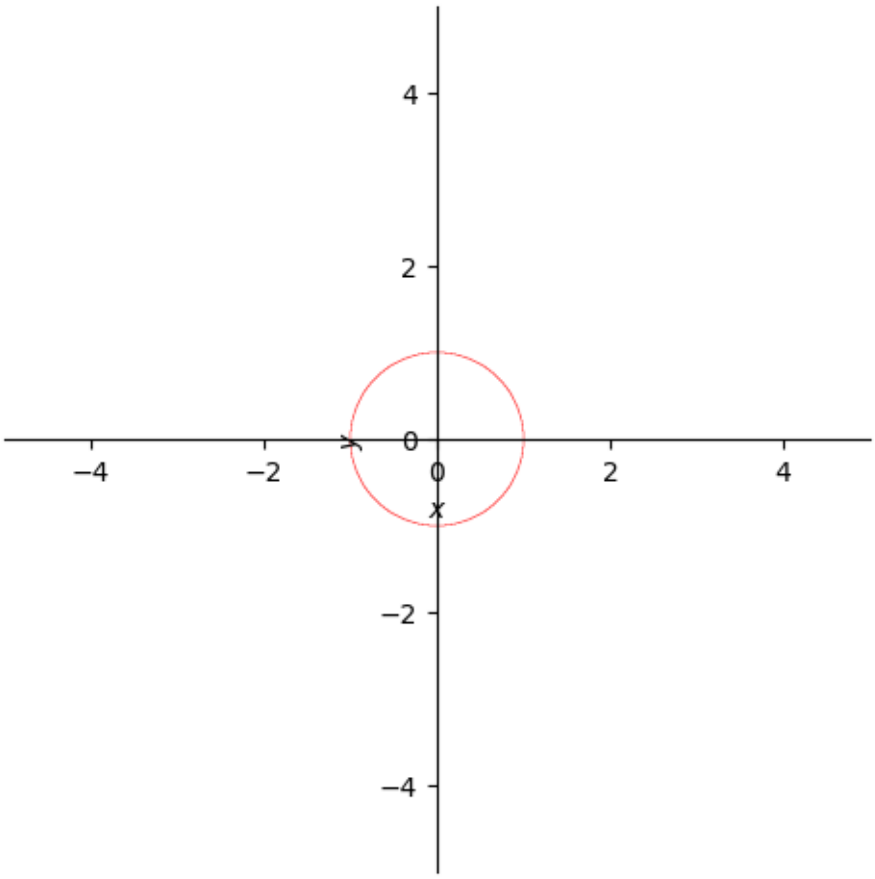
Alle Punkte (x, y) mit

$$g(x, y) = 0$$

[53]:
```
sp.Eq(x**2 - y, 0)  # Gleichung in Sympy, das mathematische =
```

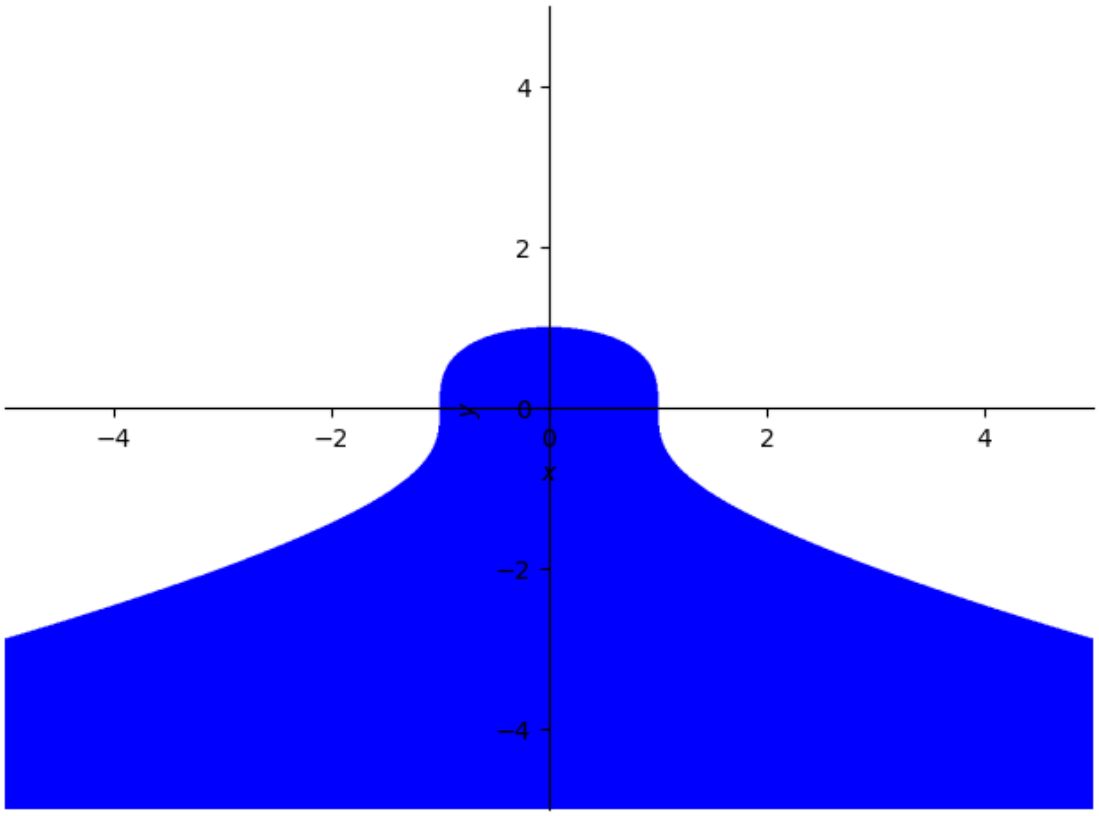
[53]:
$x^2 - y = 0$

[54]:
```
x, y, z = sp.symbols('x y z')
p3 = sp.plotting.plot_implicit(sp.Eq(x**2 + y**2, 1),
                               line_color='red',
                               aspect_ratio=(1, 1))
```



durch Ungleichung z.B. $\leq$ gegebenen Bereiche

[55]:
```
sp.plotting.plot_implicit(sp.Le(x**2+y**3, 1));
```

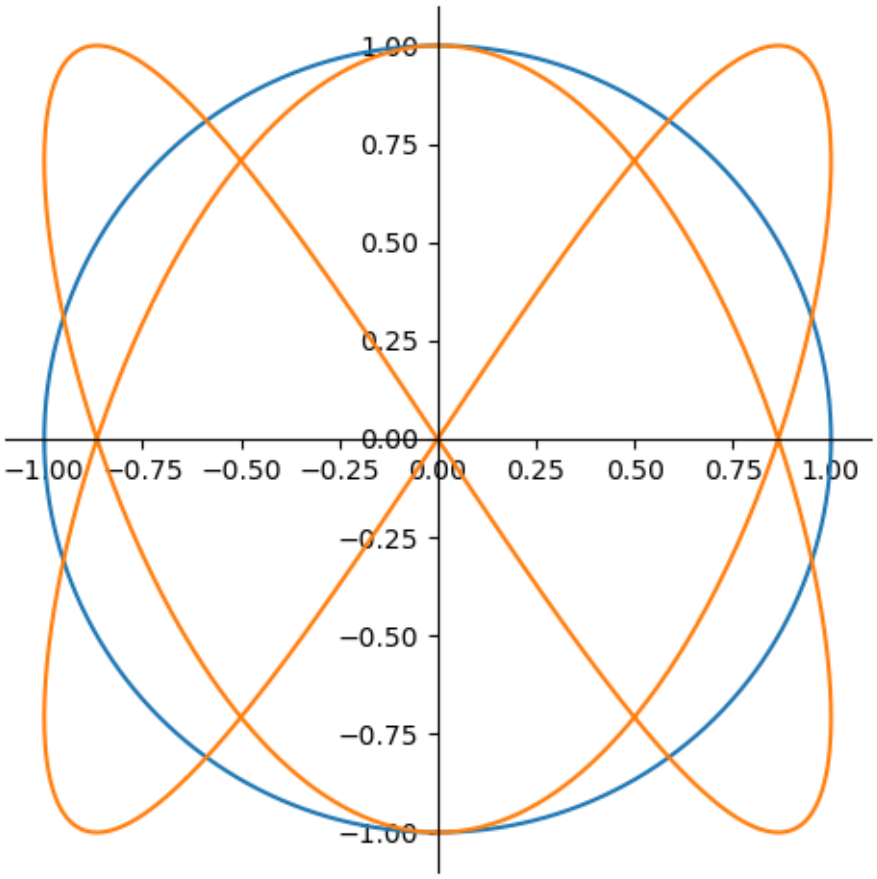


1.7.3 Kurven in Parameterdarstellung in der Ebene

Punkte (x, y) in der Ebene mit

$$x = f(t),\ y = g(t) \quad t \in [a, b]$$

[56]:
```
t = sp.symbols('t')
p1 = sp.plotting.plot_parametric((sp.sin(t), sp.cos(t)),
                                 (sin(2 * t), cos(3 * t)), (t, -sp.pi, sp.pi),
                                 aspect_ratio=(1, 1))
```



Was gibt es noch?

[57]:
```
?plotting.
```

```
Object `plotting.` not found.
```

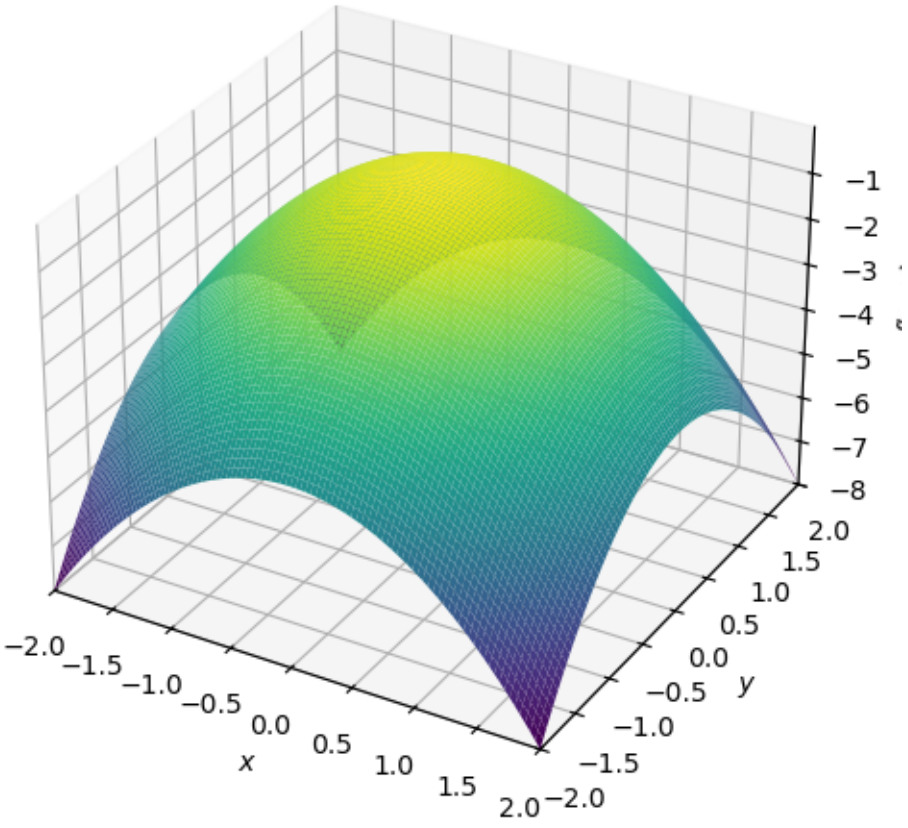
1.8 Sympy Graphik 3D

1.8.1 Flächen

als Graph einer Funktion $f : \mathbb{R}^2 \to \mathbb{R} : (x, y) \mapsto f(x, y)$

Alle Punkte $(x, y, f(x, y))$ mit $x \in (x_1, x_2),\ y \in (y_1, y_2)$

[58]:
```
sp.plotting.plot3d((-x**2 - y**2), (x, -2, 2), (y, -2, 2))
```



[58]:
```
<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at
0x7f3b02f1f860>
```

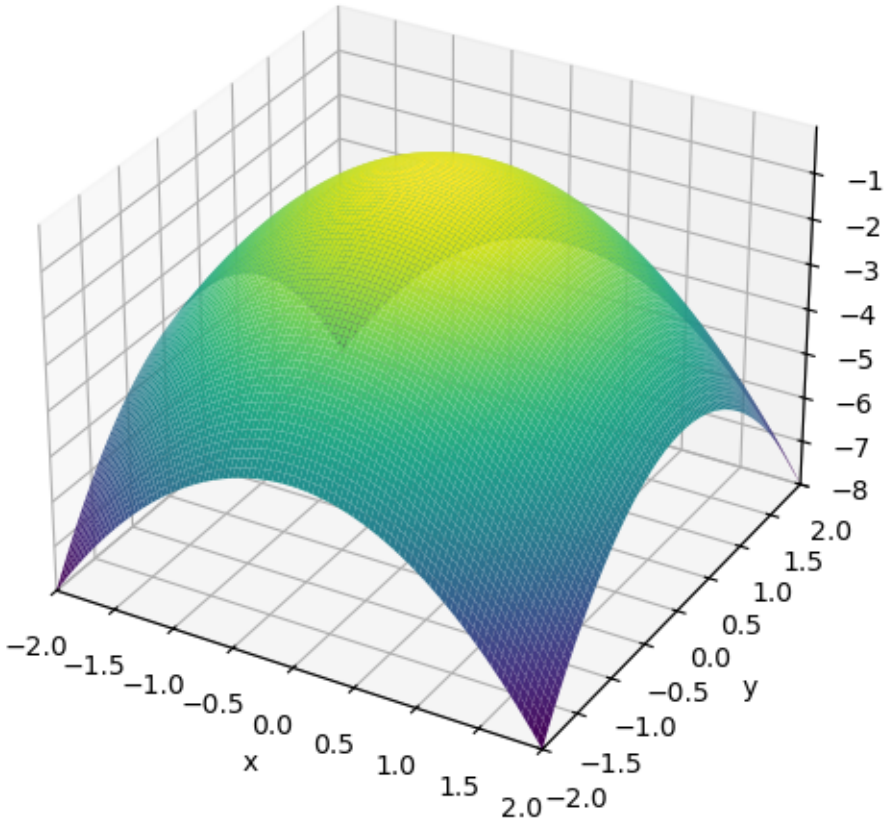
1.8.2 Parametrische Flächen

Alle Punkte $(x, y, z) \in \mathbb{R}^3$ so, dass

$$\begin{aligned} x &= f_1(u, v) \\ y &= f_2(u, v) \\ z &= f_3(u, v) \end{aligned}$$

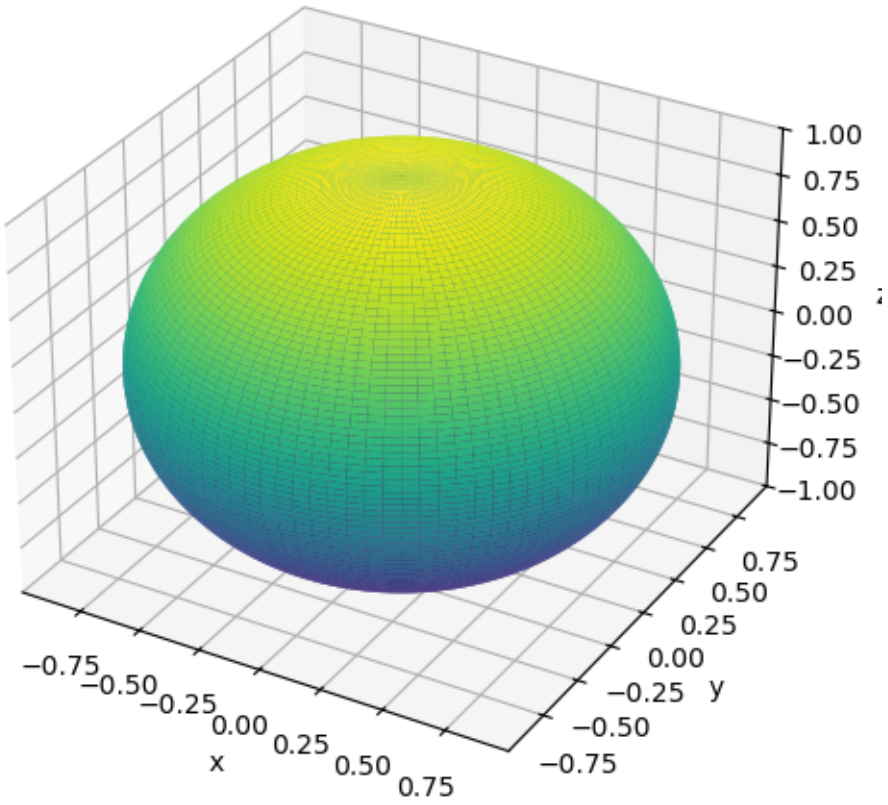
für (u, v) in einem Bereich in $\mathbb{R}^2$ und $f_j : \mathbb{R}^2 \to \mathbb{R}$.

[59]:
```
u, v = sp.symbols('u v')
sp.plotting.plot3d_parametric_surface(u, v, -u**2 - v**2, (u, -2, 2),
                                      (v, -2, 2))
```



[59]:
```
<sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at
0x7f3b0b428b00>
```

[60]:
```
azimuth, polar, radius = sp.symbols('azimuth polar radius') # Kugel
radius = 1
p4 = sp.plotting.plot3d_parametric_surface(radius*sp.sin(polar)*sp.
 ↪cos(azimuth), \
                                           radius*sp.sin(polar)*sp.
 ↪sin(azimuth), \
                                           radius*sp.cos(polar), \
                                           (polar, 0, sp.pi), (azimuth, -sp.pi,␣
 ↪sp.pi))
```



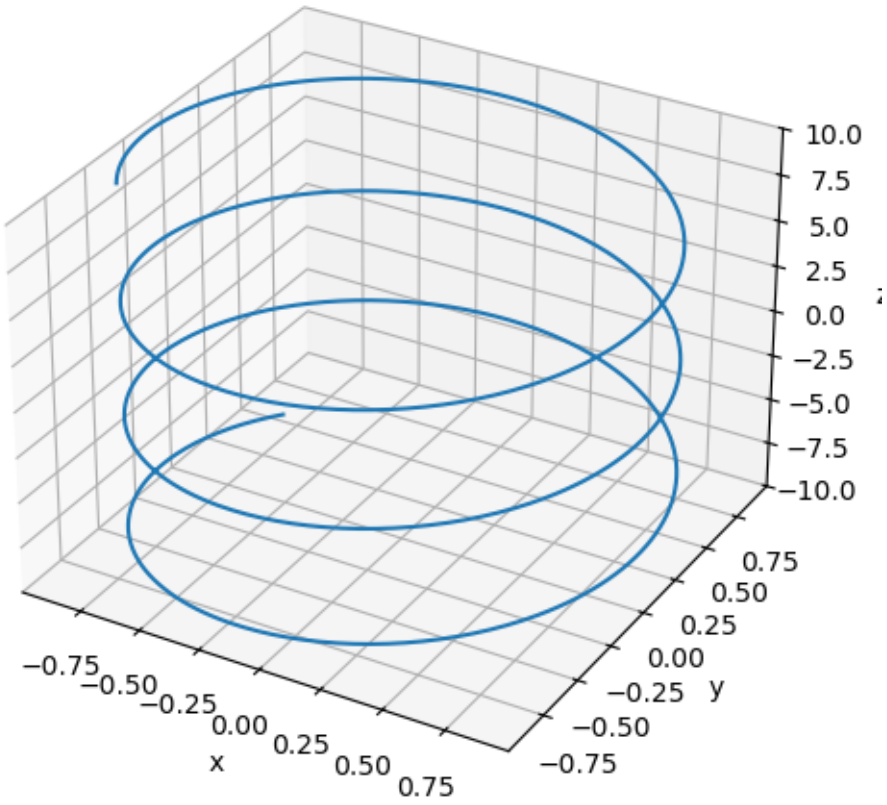
1.8.3 Kurven in Parameterdarstellung

Alle Punkte $(x, y, z) \in \mathbb{R}^3$ sodass

$$\begin{aligned} x &= f_1(u) \\ y &= f_2(u) \\ z &= f_3(u) \end{aligned}$$

für u in einem Intervall.

[61]:
```
sp.plotting.plot3d_parametric_line(sp.cos(u), sp.sin(u), u, (u, -10, 10))
```



[61]: <sympy.plotting.backends.matplotlibbackend.matplotlib.MatplotlibBackend at 0x7f3b00ea4650>