```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
# """
# Created on Mon Jan 22 10:06:38 2018
#
# @author: christianehelzel
# """
```

1 14. Vorlesung: Klassen, Vererbung

Wir betrachten Beispiele aus dem Buch von Langtangen

```
import numpy as np
import matplotlib.pyplot as plt
```

Klasse: Gerade, 1. Version:

```
class Line():
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def value(self, x):
        return self.c0 + self.c1*x
```

Die __init__ Methode wird aufgerufen, sobald ein Objekt einer Klasse initialisiert wird.

Beispiel 1

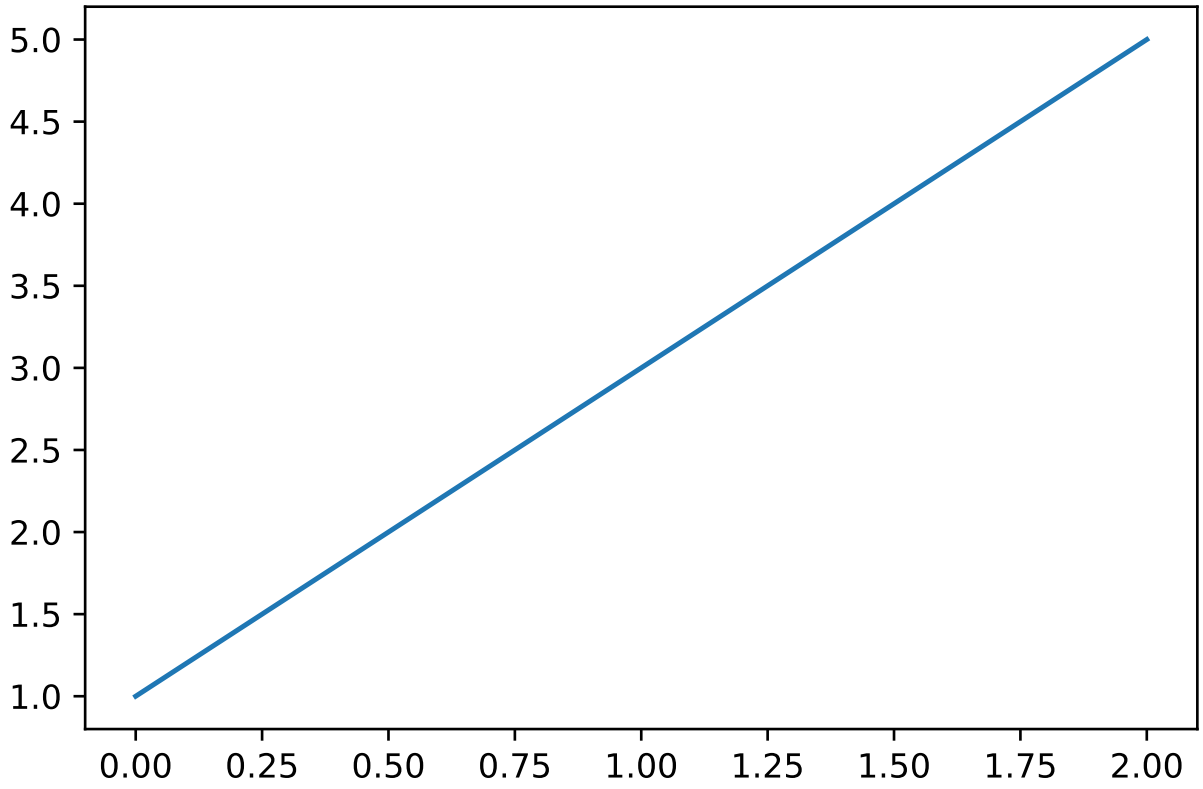
```
l = Line(1.,2.)
l.value(2.)
```

```
5.0
```

Beispiel 2

```
x = np.linspace(0.,2.,100)
plt.plot(x,l.value(x))
```

```
[<matplotlib.lines.Line2D at 0x7f6d6c0944e0>]
```



Die Call-Methode:

In obigem Beispiel wäre es praktisch, wenn man direkt l(x) statt l.value(x) schreiben kann.

Dies wird durch die __call__ Methode ermöglicht.

2. Version der Line-Klasse:

```
class Line():
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x
```

Die __init__ Methode wird bei der Initialisierung eines Objekts aufgerufen. Die __call__ Methode wird aufgerufen, wenn eine Instanz der Klasse aufgerufen wird. (Funktionsaufrufsoperator)

Beispiel

```
l = Line(1.,1.)
l(1.)
```

```
2.0
```

3. Version mit zusätzlicher Methode zur Erzeugung einer Tabelle von Funktionswerten

```
class Line():
    def __init__(self, c0, c1):
```

```
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        # abelle mit n punkten für L <= x <= R.
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s

l = Line(1.,3.)
print(l.table(0.,2.,10))
```

```
0            1
    0.222222      1.66667
    0.444444      2.33333
    0.666667            3
    0.888889      3.66667
     1.11111      4.33333
     1.33333            5
     1.55556      5.66667
     1.77778      6.33333
           2            7
```

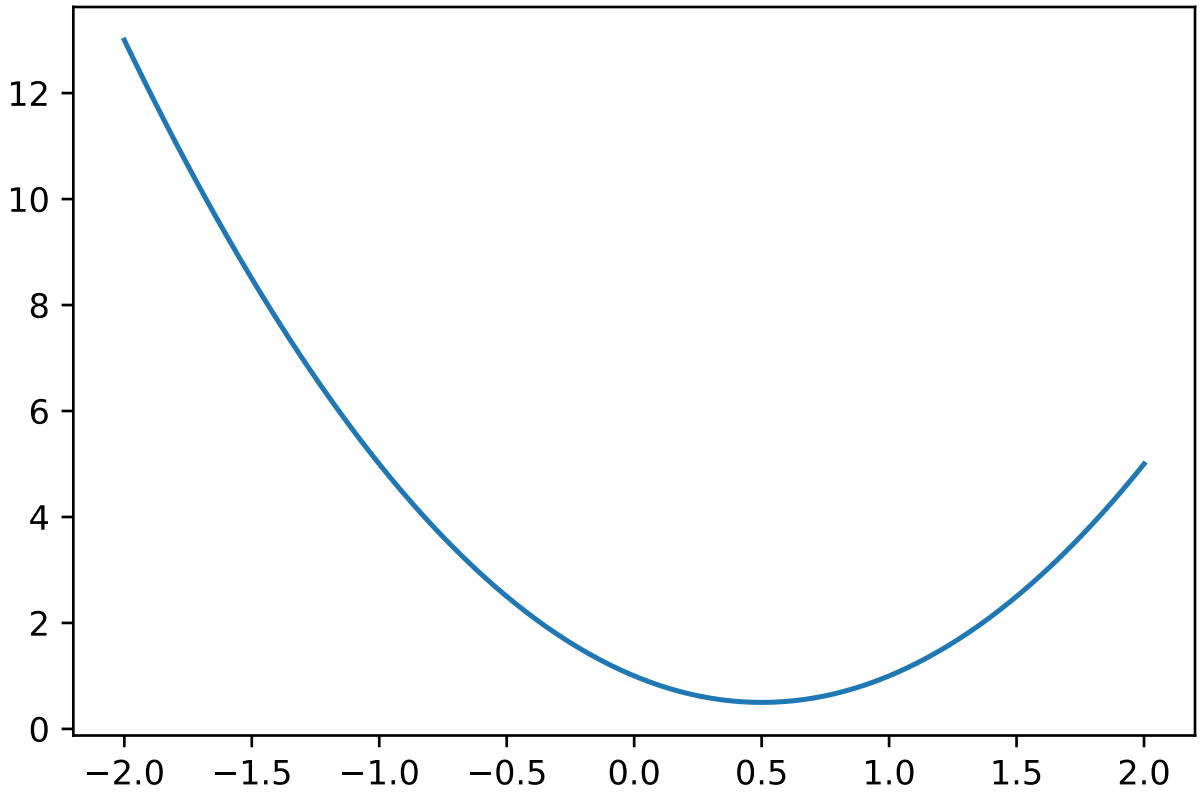
Parabel-Klasse, 1. Variante:

```
class Parabola():
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c2*x**2 + self.c1*x + self.c0

    def table(self, L, R, n):
        # Tabelle mit n Punkten für L <= x <= R.
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s
```

Beispiel

```
p = Parabola(1.,-2.,2.)
x = np.linspace(-2.,2.,100)
plt.plot(x,p(x))
```

```
[<matplotlib.lines.Line2D at 0x7f6d6c015550>]
```



1.1 Klassen-Hierarchien und Vererbung

1. Version: Parabola-Klasse, erbt von der Line-Klasse:

```
# 'Die Line-Klasse ist die Oberklasse, die Parabola-Klasse ist die Unterklasse
```

(Statt Oberklasse benutzt man auch die Synonyme Basisklasse, Elternklasse und Superklasse. Für Unterklasse gibt es auch die Synonyme abgeleitete Klasse, Kindklasse, und Subklasse.)

```
class Line():
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        # Tabelle mit n Punkten für L <= x <= R.
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s

class Parabola(Line):
    def __init__(self, c0, c1, c2):
```

```
        Line.__init__(self,c0, c1) # Line initialisiert c0 und c1
        self.c2 = c2

    def __call__(self, x):
        return Line.__call__(self, x) + self.c2*x**2
```

Die Parabola-Klasse erbt die table-Methode von Line

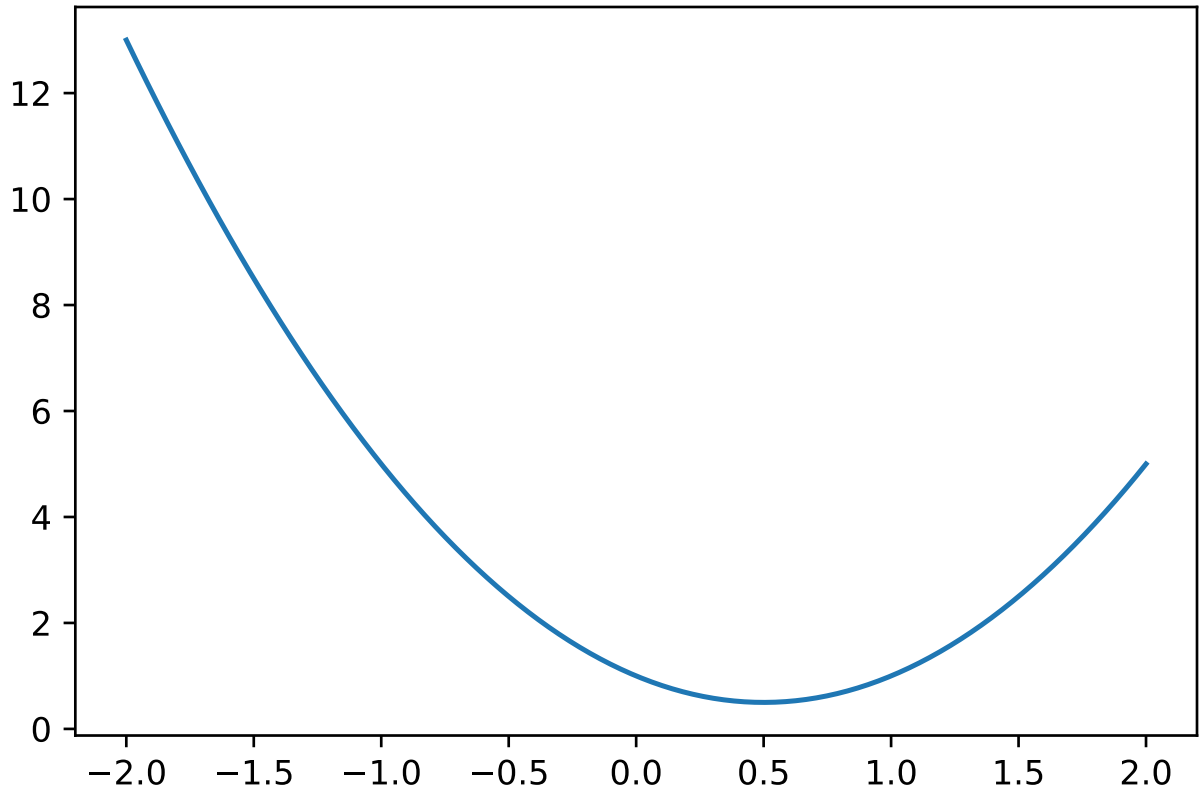
```
p = Parabola(1., -2., 2.)
p(2.5)

p.c0
p.c1
p.c2
```

```
2.0
```

Beispiel

```
x = np.linspace(-2.,2.,100)
plt.plot(x,p(x))
print(p.table(0.,1.,10))
```

```
0            1
    0.111111     0.802469
    0.222222     0.654321
    0.333333     0.555556
    0.444444     0.506173
    0.555556     0.506173
    0.666667     0.555556
    0.777778     0.654321
    0.888889     0.802469
           1            1
```



Überprüfe den Klassen-Typ

Die Python Funktion isinstance(i,t) überprüft, ob eine Instanz i den Klassen-Typ t hat.

```
l = Line(-1,1)
isinstance(l,Line)
```

```
True
```

und

```
isinstance(l,Parabola)
```

```
False
```

Eine Gerade ist keine Parabel

```
p = Parabola(-1,0,10)
isinstance(p,Parabola)
```

```
True
```

Ist eine Parabel eine Gerade?

```
isinstance(p,Line)
```

```
True
```

Ja, bezüglich der Klassen-Hierarchie ist eine Parabel eine Gerade.

Die Python Funktion issubclass(c1, c2) testet ob die Klasse c1 eine Unterklasse der Klasse c2 ist.

```
issubclass(Parabola,Line)
```

```
True
```

und

```
issubclass(Line,Parabola)
```

```
False
```

1.2 Attribute vs. Vererbung

Die Parabola-Klasse kann auch eine Instanz der Line-Klasse als Attribut enthalten.

```
class Line():
    def __init__(self, c0, c1):
        self.c0 = c0
        self.c1 = c1

    def __call__(self, x):
        return self.c0 + self.c1*x

    def table(self, L, R, n):
        # Tabelle mit n Punkten für L <= x <= R.
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s

class Parabola(Line):
    def __init__(self, c0, c1, c2):
        self.line = Line(c0, c1) # line speichert c0 und c1
        self.c2 = c2

    def __call__(self, x):
        return self.line(x) + self.c2*x**2

l = Line(1.,1.)
l(1.5)
```

```
2.5
```

und

```
p = Parabola(1.,1.,1.)
p(2.3)
```

```
8.59
```

Beispiel

```
p.c2
p.line # Objekt der Klasse Line
```

```
<__main__.Line at 0x7f6d60e2b9e8>
```

geht nicht

```
p.c0
```

```
---------------------------------------------------------------------------AttributeErro
call last)<ipython-input-1-6430950050e9> in
<module>()
----> 1p.c0
AttributeError: 'Parabola' object has no
attribute 'c0'
```

geht nicht

```
p.c1
```

```
---------------------------------------------------------------------------AttributeErro
call last)<ipython-input-1-265981771e25> in
<module>()
----> 1p.c1
AttributeError: 'Parabola' object has no
attribute 'c1'
```

geht wieder

```
p.line.c0
p.line.c1
p.line.table(0.,1.,10)
```

```
'           0            1    0.111111      1.11111    0.222222
1.22222    0.333333      1.33333    0.444444      1.44444
0.555556      1.55556    0.666667      1.66667    0.777778
1.77778    0.888889      1.88889           1            2'
```

Für Mathematiker ist eine Parabel keine Gerade aber eine Gerade ist ein Spezialfall einer Parabel. Dies können wir folgendermaßen implementieren: Die Line-Klasse erbt von der Parabola-Klasse Line ist jetzt eine Unterklasse von Parabola

```
class Parabola():
    def __init__(self, c0, c1, c2):
        self.c0 = c0
        self.c1 = c1
        self.c2 = c2

    def __call__(self, x):
        return self.c0 + self.c1*x + self.c2*x**2

    def table(self, L, R, n): # implemented as shown above
        # Tabelle mit n Punkten für L <= x <= R.
        s = ''
        import numpy as np
        for x in np.linspace(L, R, n):
            y = self(x)
            s += '%12g %12g\n' % (x, y)
        return s

class Line(Parabola):
    def __init__(self, c0, c1):
        Parabola.__init__(self, c0, c1, 0)

l = Line(1.,1.)  # Es werden zwei Parameter übergeben
l(2.)

l.c0
l.c1
l.c2

p = Parabola(1.,1.,1.)
p(4.)
```

```
21.0
```

1.3 Weitere Klassen zur Representation von Funktionen

Neben der __call__ Methode sollen Methoden zur Berechnung der 1. und 2. Ableitung bereitgestellt werden

Superclass: Enthält numerische Methoden zur Berechnung der Ableitungen die vererbt werden

```
class FuncWithDerivatives():
    def __init__(self, h=1.0E-5):
        self.h = h

    def __call__(self, x):
        raise NotImplementedError\
        ('___call__ missing in class %s' % self.__class__.__name__)

    def df(self, x):
        # Liefert Approximation an 1. Ableitung
        # Compute first derivative by a finite difference
        h = self.h
        return (self(x+h) - self(x-h))/(2.0*h)

    def ddf(self, x):
```

```
        # Return the 2nd derivative of self.f.
        # Compute second derivative by a finite difference:
        h = self.h
        return (self(x+h) - 2*self(x) + self(x-h))/(float(h)**2)
```

Eine konkrete Funktion ist eine Unterklasse von FuncWithDerivatives

f(x) = cos(ax) + x^3

```
class MyFunc(FuncWithDerivatives):
    import numpy as np

    def __init__(self,a):
        self.a = a

    def __call__(self,x):
        return np.cos(self.a*x) + x**3

    def df(self, x):
        a = self.a
        return -a*np.sin(a*x) + 3*x**2

    def ddf(self, x):
        a = self.a
        return -a*a*np.cos(a*x) + 6*x
```

Beachte, dass die übergeordnete Klasse FuncWithDerivatives nicht aufgerufen wird.

```
f = MyFunc(1.)
f.df(2.)
f.ddf(2.)
```

```
12.416146836547142
```

f(x) = ln |p tanh(q x cos(rx))|

Für diese Funktion verzichten wir darauf, die Ableitungen explizit zu berechnen. Stattdessen verwenden wir die Approximationen, die in der Oberklasse implementiert sind.

```
class MyComplicatedFunc(FuncWithDerivatives):
    import numpy as np

    def __init__(self, p, q, r):
        FuncWithDerivatives.__init__(self)
        self.p = p
        self.q = q
        self.r = r

    def __call__(self, x):
        return np.log(abs(self.p*np.tanh(self.q*x*np.cos(self.r*x))))


f = MyComplicatedFunc(1.,1.,1.)
print(f(1.))
print(f.df(1.))
print(f.ddf(1.))
```

```
-0.7068065121515873
-0.46207988960156315
-3.7618941295392
```