**Solutions to Exercises to**

# Programming Methods in Scientific Computing

David Bauer
Jendrik Stelzner

Letzte Änderung: 22. November 2017

# Chapter 3 Python, the Fundamentals

## Exercise 3.2

Wir weitern die gegeben Klasse `Polynomial` um eine Methode `derivative` zum Ableiten, sowie eine Methode `antiderivative` zum Bilden einer Stammfunktion. Dabei wählen wir die „Integrationskonstante“ als 0. Wir definieren außerdem Funktionen zum Ausgeben von Polynomen durch die `print`-Funktion.

```
1  class Polynomial:
2      def __init__(self, coeff):
3          self.coeff = coeff
4
5      def __str__(self):
6          s = ""
7          if self.coeff == []:
8              s = "0"
9          else:
10             s += "{} x^0".format(self.coeff[0])
11         for i in range(1,len(self.coeff)):
12             s += " + {} x^{}".format(self.coeff[i], i)
13         return s
14
15     def __repr__(self):
16         return str(s)
17
18     def __call__(self, x):
19         s = 0
20         for i in range(len(self.coeff)):
21             s += self.coeff[i]*x**i
22         return s
23
24     def __add__(self, other):
25         l = []
26         if len(self.coeff) > len(other.coeff):
27             l += self.coeff
28             for i in range(len(other.coeff)):
29                 l[i] += other.coeff[i]
30         else:
31             l += other.coeff
32             for i in range(len(self.coeff)):
33                 l[i] += self.coeff[i]
34         return Polynomial(l)
35
36     def __eq__(self, other):
```

Für das gegebene Polynom $p(x) = 3x^2 + 2x + 1$ testen wir unser Programm mit dem folgenden Code:

```
p = Polynomial([1,2,3])
print("The given polynmial p:")
print(p)
print("The derivative of p:")
print( p.derivative() )
print("The antiderivative of p:")
print( p.antiderivative() )
print("Taking antiderivative and then derivative:")
```

Dabei erhalten wir den folgenden Output:

```
$ python exercise_03_02.py
The given polynmial p:
1 x^0 + 2 x^1 + 3 x^2
The derivative of p:
2 x^0 + 6 x^1
The antiderivative of p:
0 x^0 + 1.0 x^1 + 1.0 x^2 + 1.0 x^3
Taking antiderivative and then derivative:
1.0 x^0 + 2.0 x^1 + 3.0 x^2
```

## Exercise 3.3

Wir definieren direkt Klasse `Matrix`, die über alle Methoden verfügt, die wir in diesem und den späteren Aufgabenteilen nutzen werden.

```
class Matrix():
    def __init__(self, entries):
        m = len(entries)
        if m == 0:
            raise ValueError("height must be positive")
        n = len(entries[0])
        if n == 0:
            raise ValueError("width must be positive")
        for i in range(1, m):
            if len(entries[i]) != n:
                raise ValueError("rows must have the same width")
        self.height = m
        self.width = n
        self.entries = entries

    def __getitem__(self, i):       # allows to get the rows via A[i]
        return self.entries[i]

    def __setitem__(self, i, k):    # allows to set rows via A[i]
        self.entries[i] = k

    def __str__(self):              # allows print(A) for a Matrix A
        rows = ["["]*self.height
        for j in range(self.width): # construct output columnwise, align left
            numbers = []            # numbers to appear in column j
            maxlen = 0              # maximal length of a number in column j
            for i in range(self.height):
```

```
                s = str(self[i][j])
                numbers.append(s)
                if len(s) > maxlen:
                    maxlen = len(s)
            for i in range(self.height):
                # pad the entries if they are too short
                rows[i] += numbers[i] + " "*(maxlen-len(numbers[i])) + " "
        s = ""
        for r in rows:
            s += r[:-1] + "]\n" # remove white space at the end of ech line
        s = s[:-1]              # remove empty line at the end
        return s

    def __repr__(self):
        return str(self)

    def __mul__(self, other):
        if self.width != other.height:
            raise TypeError('matrix dimensions do not match')
        newentries = []
        for i in range(self.height):
            row = []
            for j in range(other.width):
                s = self[i][0] * other[0][j]    # makes s have the right type
                for k in range(1, self.width):
                    s += self[i][k] * other[k][j]
                row.append(s)
            newentries.append(row)
        return Matrix(newentries)

    def __eq__(self, other):
        if self.height != other.height or self.width != other.width:
                return False
        for i in range(self.height):
            for j in range(self.width):
                if self[i][j] != other[i][j]:
                    return False
        return True

    def mapentries(self, f):        # applies a function to all entries
        A = zeromatrix(self.height, self.width) # zeromatrix is defined below
        for i in range(self.height):
            for j in range(self.width):
                A[i][j] = f(self[i][j])
        return A

    def addrow(self, i, j, c):      # add c times row j to row i
        for k in range(self.width):
            self[i][k] =  c * self[j][k] + self[i][k]
            # makes c responsible for implementing the operations

    def addcolumn(self, i, j, c):   # add c times column j to column i
        for k in range(self.height):
            self[k][i] = c * self[k][j] + self[k][i]

    def multrow(self, i, c):        # multiply row i with c
```

```
84          for j in range(self.width):
85              self[i][j] = c * self[i][j]
86
87      def multcolumn(self, j, c):     # multiply row j with c
88          for i in range(self.height):
89              self[i][j] = c * self[i][j]
90
91      def swaprows(self, i, j):       # swap rows i and j
92          if i > self.height or j > self.height:
93              raise ValueError("swap nonexistent rows")
94          l = self[i]
95          self[i] = self[j]
96          self[j] = l
97
98      def transpose(self):
99          T = zeromatrix(self.width, self.height)
100         for i in range(self.height):
101             for j in range(self.width):
102                 T[j][i] = self[i][j]
103         return T
```

Wir definieren zudem Hilfsfunktionen, die wir im Weiteren nutzen werden:

```
1  def zeromatrix(height, width):  # creates a zero matrix
2      entries = []
3      for i in range(height):
4          entries.append([0]*width)
5      return Matrix(entries)
6
7  def identitymatrix(size):       # creates an identiy matrix
8      E = zeromatrix(size, size)
9      for i in range(size):
10         E[i][i] = 1
11     return E
12
13 def copymatrix(A):              # forcefully copies a matrix
14     B = zeromatrix(A.height, A.width)
15     for i in range(A.height):
16         for j in range(A.width):
17             B[i][j] = A[i][j]
18     return B
```

Die Assoziativität der Matrixmultiplikation testen wir mit dem folgenden Code:

```
1  A = Matrix([[0,1],[1,0],[1,1]])
2  print("A:")
3  print(A)
4  B = Matrix([[1,2,3,4],[5,6,7,8]])
5  print("B:")
6  print(B)
7  C = Matrix([[1,0],[0,1],[1,0],[0,1]])
8  print("C:")
9  print(C)
10 print("Checking if A(BC) == (AB)C:")
11 print(A * (B * C) == (A * B) * C)
```

Wir erhalten wir (durch Ausführen in der Konsole) den folgenden Output:

```
$ python exercise_03_03.py
A:
[0 1]
[1 0]
[1 1]
B:
[1 2 3 4]
[5 6 7 8]
C:
[1 0]
[0 1]
[1 0]
[0 1]
Checking if A(BC) == (AB)C:
True
```

## Exercise 3.4

Wir schreiben zunächst eine Klasse `Rational`, die ein genaues Rechnen mit rationalen Zahlen erlaubt.

```
1  class Rational():
2      def __init__(self, num, denum = 1):   # default denumerator is 1
3          if type(num) == Rational:
4              p = num/Rational(denum)
5              self.num = p.num
6              self.denum = p.denum
7          elif type(num) != int:
8              raise TypeError("numerator is no integer")
9          elif type(denum) != int:
10             raise TypeError("denumerater is no integer")
11         elif denum == 0:
12             raise ZeroDivisionError("denumerator is zero")
13         else:
14             self.num = num
15             self.denum = denum
16
17     def __str__(self):    # allows print(x) for Rational x
18         return "{}/{}".format(self.num, self.denum)
19
20     def __repr__(self):
21         return str(self)
22
23     def __add__(self, other):
24         if type(other) == int:
25             return self + Rational(other)
26         elif type(other) == Rational:
27             return Rational( self.num * other.denum + self.denum * other.num,
                   self.denum * other.denum )
28         raise TypeError("unsupported operand type(s) for + or add(): '
               Rational' and '{}'".format(type(other).__name__))
29
30     def __sub__(self, other):
```

```
31          if type(other) == int:
32              return self - Rational(other)
33          elif type(other) == Rational:
34              return Rational( self.num * other.denum - self.denum * other.num,
                    self.denum * other.num )
35          raise TypeError("unsupported operand type(s) for - or sub(): '
                Rational' and '{}'".format(type(other).__name__))
36
37      def __mul__(self, other):
38          if type(other) == int:
39              return self * Rational(other)
40          elif type(other) == Rational:
41              return Rational( self.num * other.num, self.denum * other.denum )
42          raise TypeError("unsupported operand type(s) for * or mul(): '
                Rational' and '{}'".format(type(other).__name__))
43
44      def __truediv__(self, other):
45          if type(other) == int:
46              return self / Rational(other)
47          elif type(other) == Rational:
48              if other.num == 0:
49                  raise ZeroDivisionError("division by zero")
50              return Rational( self.num * other.denum, self.denum * other.num)
51          raise TypeError("unsupported operand type(s) for / or truediv(): '
                Rational' and '{}'".format(type(other).__name__))
52
53      def __pow__(self, n):   # supports only integer powers
54          if not type(n) == int:
55              raise TypeError("unsupported operand type(s) for ** or pow(): '
                    Rational' and '{}'".format(type(n).__name__))
56          if n >= 0:
57              return Rational(self.num**n, self.denum**n)
58          return Rational(self.denum, self.num)**(-n)
59
60      def __pos__(self):
61          return Rational( self.num, self.denum )
62
63      def __neg__(self):
64          return Rational( -self.num, self.denum )
65
66      def __abs__(self):
67          if self.num <= 0 and self.denum > 0:
68              return Rational(-self.num, self.denum)
69          elif self.num >= 0 and self.denum < 0:
70              return Rational(self.num, -self.denum)
71          return Rational(self.num, self.denum)
72
73      def __eq__(self, other):
74          if type(other) == int:  # allows comparison to int, used for x == 0
75              return (self == Rational(other))
76          return (self.num * other.denum == self.denum * other.num)
77
78      def __float__(self):
79          return self.num / self.denum
```

Die LU-Zerlegung von passenden Matrix mit ganzzahligen Einträgen kann nun dem

folgenden naiven Algorithmus berechnet werden:

```
def naive_lu(A):  # expects a Rational valued matrix as input
    if A.height != A.width:
        raise ValueError("matrix is not square")
    U = copymatrix(A)   # circumvent pass by reference
    n = U.height
    L = identitymatrix(n)
    # bring U in upper triangular form, change L such that always LU = A
    for j in range(n):
        if U[j][j] == 0:
            raise ValueError("algorithm does not work for this matrix")
        for i in range(j+1,n):
            L.addcolumn(j, i,  U[i][j]/U[j][j]) # important: change L first
            U.addrow(i, j, -U[i][j]/U[j][j])
    return (L,U)
```

Wir testen das Program anhand der gegebenen Matrizen

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 6 & 3 \\ 9 & 10 & 6 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

mit dem folgenden Code:

```
A = Matrix([[3,2,1],[6,6,3],[9,10,6]])
print("A:")
print(A)
(L,U) = naive_lu(A.mapentries(Rational))
print("L:")
print(L)
print("U:")
print(U)
B = Matrix([[0,1],[1,0]])
print("B:")
print(B)
print("Trying to calculate the LU decomposition of B:")
(L,U) == naive_lu(B)
```

Als Output erhalten wir folgendes:

```
$ python exercise_03_04.py
A:
[3 2  1]
[6 6  3]
[9 10 6]
L:
[9/9  0/18  0]
[18/9 18/18 0]
[27/9 36/18 1]
U:
[3/1   2/1   1/1    ]
[0/3   6/3   3/3    ]
[0/162 0/162 162/162]
Check if L*U == A:
True
```

```
B:
[0 1]
[1 0]
Trying to calculate the LU decomposition of B:
Traceback (most recent call last):
  File "exercise_03_04.py", line 62, in <module>
    (L,U) == naive_lu(B)
  File "exercise_03_04.py", line 15, in naive_lu
    raise ValueError("algorithm does not work for this matrix")
ValueError: algorithm does not work for this matrix
```

Da die Matrix $B$ keine LU-Zerlegung besitzt, ist es okay, dass unser Algorithmus diese nicht findet.

## Exercise 3.5

Wir bestimmen die Cholesky-Zerlegung eintragsweise.

```
1  from math import sqrt
2
3  def cholesky(A):    # expects int or float as matrix entries
4      if A.height != A.width:
5          raise ValueError("matrix is not square")
6      B = copymatrix(A)
7      n = B.height
8      L = zeromatrix(n,n)
9      for i in range(n):
10         rowsum = 0
11         for j in range(i):
12             s = 0
13             for k in range(j):
14                 s += L[i][k] * L[j][k]
15             L[i][j] = (B[i][j] - s)/L[j][j]
16             rowsum += L[i][j]**2
17         L[i][i] = sqrt( B[i][i] - rowsum )
18     return L
```

Für die gegebenen Matrizen

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 2 & 5 & 2 \\ 1 & 2 & 10 \end{pmatrix} \quad \text{und} \quad B = \begin{pmatrix} 1.01 \cdot 10^{-2} & 0.705 & 1.42 \cdot 10^{-2} \\ 0.705 & 49.5 & 1 \\ 1.42 \cdot 10^{-2} & 1 & 1 \end{pmatrix}$$

testen wir unser Programm mit dem folgenden Code:

```
1  A = Matrix([[1,2,1],[2,5,2],[1,2,10]])
2  print("A:")
3  print(A)
4  L = cholesky(A)
5  print("L:")
6  print(L)
7  print("L * L^T:")
8  print(L * L.transpose())
```

```
B = Matrix([[1.01E-2, 0.705, 1.42E-2],[0.705,49.5,1],[1.42E-2,1,1]])
print("B:")
print(B)
L = cholesky(B)
print("L:")
print(L)
print("L * L^T:")
print(L * L.transpose())
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_05.py
A:
[1 2 1 ]
[2 5 2 ]
[1 2 10]
L:
[1.0 0   0  ]
[2.0 1.0 0  ]
[1.0 0.0 3.0]
L * L^T:
[1.0 2.0 1.0 ]
[2.0 5.0 2.0 ]
[1.0 2.0 10.0]
B:
[0.0101 0.705 0.0142]
[0.705  49.5  1     ]
[0.0142 1     1     ]
L:
[0.1004987562112089  0                    0                 ]
[7.015012190980423   0.5381486415443629   0                 ]
[0.14129528100981847 0.016374437298272527 0.9898320672556135]
L * L^T:
[0.010100000000000001 0.705 0.014200000000000003]
[0.705                 49.5  1.0                 ]
[0.014200000000000003 1.0   1.0                 ]
```

## Exercise 3.6

Mithilfe elementarer Zeilenumformungen, die in der Klasse `Matrix` implementiert sind, lässt sich nun der Gauß-Algorithmus zum Invertieren von Matrizen implementieren.

```
# we assume that the input matrix is well-behaved
# i.e. entries allow a reasonable comparison to 0
def invert(A):
    if A.height != A.width:
        raise ValueError("matrix is not square")
    B = copymatrix(A)   # circumvent pass by reference
    n = B.height
    Inv = identitymatrix(n)
    B = B.mapentries(Rational)      # make all
    Inv = Inv.mapentries(Rational)  # entries rational
    # bring B in lower triangular form
    for j in range(n):
```

```
13          p = -1
14          for i in range(j,n):
15              if B[i][j] != 0:
16                  p = i
17                  break
18          if p == -1:
19              raise ZeroDivisionError("matrix is not invertible")
20          for i in range(p+1,n):
21              Inv.addrow(i, p, -B[i][j]/B[p][j])  # import: change inverse
                    first
22              B.addrow(i, p, -B[i][j]/B[p][j])
```

Wir testen unser Programm anhand der gegebenen Matrix

$$A = \begin{pmatrix} 3 & -1 & 2 \\ -3 & 4 & -1 \\ -6 & 5 & -2 \end{pmatrix}$$

mit dem folgenden Code:

```
1  A = Matrix([[3,-1,2],[-3,4,-1],[-6,5,-2]])
2  print("A:")
3  print(A)
4  B = invert(A)
5  print("A^(-1) with rationals:")
6  print(B)
7  print("A^(-1) with floats:")
8  print(B.mapentries(float))
9  print("Checking if A*B = I (using rationals):")
10 print(A.mapentries(Rational) * B == identitymatrix(3))
```

Dabei nutzen wir erneut die Klasse `Rational`, um ein genaues Rechnen zu erlauben. Wir erhalten den folgenden Output:

```
python exercise_03_06.py
A:
[3  -1 2 ]
[-3 4  -1]
[-6 5  -2]
A^(-1) with rationals:
[-1162261467/3486784401 3099363912/3486784401 -2711943423/3486784401]
[0/43046721             28697814/43046721     -14348907/43046721    ]
[59049/59049            -59049/59049          59049/59049           ]
A^(-1) with floats:
[-0.3333333333333333 0.8888888888888888 -0.7777777777777778]
[0.0                 0.6666666666666666 -0.3333333333333333]
[1.0                 -1.0               1.0                ]
Checking if A*B = I (using rationals):
True
```

## Exercise 3.7

Wir Berechnen die QR-Zerlegung einer nicht-singulären Matrix $A$ durch Anwenden des Gram-Schmidt-Verfahrens auf die Spalten von $A$, von links nach rechts:

```
from math import sqrt

# assumes the matrix to have integer or float values
# and to be nonsingular
def qrdecomp(A):
    if A.height != A.width:
        return ValueError("only square matrices are supported")
    n = A.height
    Q = copymatrix(A)
    R = identitymatrix(n)
    for j in range(n):
        # make the j-th column of Q orthogonal to the next columns
        for k in range(j):
            s = 0   # inner product of j-th and k-th columns
            for i in range(n):
                s += Q[i][j] * Q[i][k]
            Q.addcolumn(j, k, -s)
            R.addrow(k, j, s)
        sn = 0   # squared norm of the j-th column
        for i in range(n):
            sn += Q[i][j]**2
        Q.multcolumn(j, sqrt(sn)**(-1))
        R.multrow(j, sqrt(sn))
    return (Q,R)
```

Für die gegebene Matrix

$$A = \begin{pmatrix} 12 & -51 & 4 \\ 6 & 167 & -68 \\ -4 & 24 & -41 \end{pmatrix}$$

testen wir das Programm mithilfe des folgenden Codes:

```
A = Matrix([[12,-51,4],[6,167,-68],[-4,24,-41]])
print("A:")
print(A)
(Q,R) = qrdecomp(A)
print("Q:")
print(Q)
print("Q * Q^T:")
print(Q * Q.transpose())
print("R:")
print(R)
print("Q*R:")
print(Q*R)
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_07.py
A:
[12 -51 4  ]
[6  167 -68]
[-4 24  -41]
Q:
[0.8571428571428571  -0.3942857142857143 -0.33142857142857124]
```

```
[0.42857142857142855 0.9028571428571428  0.03428571428571376 ]
[-0.2857142857142857 0.17142857142857143 -0.9428571428571428 ]
Q * Q^T:
[0.9999999999999998      1.474514954580286e-16 -1.6653345369377348e-16]
[1.474514954580286e-16   0.9999999999999998    4.996003610813204e-16  ]
[-1.6653345369377348e-16 4.996003610813204e-16 1.0                    ]
R:
[14.0 20.999999999999996 -14.000000000000002]
[0.0  175.0              -69.99999999999999 ]
[0.0  0.0                35.0               ]
Q*R:
[12.0 -51.0 4.000000000000002]
[6.0  167.0 -68.0            ]
[-4.0 24.0  -41.0            ]
```

# Exercise 3.8

## (1)

Alle notwendigen Funktionswerte werden zunächst berechnet und in einer Liste gespeichert, um mehrfaches Berechnen des gleichen Funktionswertes zu umgehen.

```
1 def trapeze(f,a,b,n):
2     values = [f(a + (k/n)*(b-a)) for k in range(n+1)]
3     integral = 0
4     for i in range(len(values)-1):
5         integral += values[i] + values[i+1]
6     integral = (b-a)*integral/n/2
7     return integral
```

## (2)

Wir testen unser Programm anhand des gegebenen Integrals $\int_0^\pi \sin(x)\,\mathrm{d}x$ mit dem folgenden Code:

```
1 from math import sin, pi
2 n = 1
3 s = 0
4 while 2 - s >= 1.E-6:   # sin is concave on [0,pi] -> estimate too small
5     n += 1              # can skip n = 1 because it results in 0
6     s = trapeze(sin, 0, pi, n)
7
8 print("Estimate for integral of sin from 0 to pi using trapeze:")
9 print(s)
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_08.py
Estimate for integral of sin from 0 to pi using trapeze:
1.9999990007015205
```

# Exercise 3.9

## (1)

Wir definieren eine neue Funktion `powertrapeze`, welche das angegebene Verfahren implementiert:

```
def powertrapeze(f, a, b, mmax):
    integrals = []              # list of the approximations
    values = [f(a), f(b)]       # list of the calculated values
    for m in range(1,mmax+1):
        n = 2**m
        for k in range(1,n,2):
            values.insert(k, f(a + (k/n)*(b-a)))    # add new values
        integral = 0
        for i in range(len(values)-1):
            integral += values[i] + values[i+1]
        integral = (b-a)*integral/n/2
        integrals.append(integral)                  # add new approx.
    return integrals
```

Hiermit berechnen die Approximationen für $\int_0^\pi \sin(x)\,\mathrm{d}x$ für $m = 1, \ldots, 10$ mit dem folgenden Code:

```
from math import sin, pi
m = 10
results = powertrapeze(sin, 0, pi, m)
print("Calculate trapeze estimate for int. of sin from 0 to pi, 2^m intervals
    :")
print(" m \testimate \t\terror")
for i in range(m):
    print("{:2d}\t{}\t{:.20f}".format(i+1, results[i], 2-results[i]))
```

Wir erhalten den folgenden Output:

```
Calculate trapeze estimate for int. of sin from 0 to pi, 2^m intervals:
 m      estimate               error
 1      1.5707963267948966     0.42920367320510344200
 2      1.8961188979370398     0.10388110206296019555
 3      1.9742316019455508     0.02576839805444919307
 4      1.9935703437723395     0.00642965622766045186
 5      1.9983933609701445     0.00160663902985547224
 6      1.9995983886400386     0.00040161135996141795
 7      1.9998996001842035     0.00010039981579645918
 8      1.9999749002350518     0.00002509976494824429
 9      1.9999937250705773     0.00000627492942273378
10      1.9999984312683816     0.00000156873161838433
```

## (2)

Es fällt auf, dass sich der Fehler in jedem Schritt etwa geviertelt wird. Bezeichnet $a_n$ die $n$-te Approximation, so gilt $a_0 \leq a_1 \leq \cdots \leq a_n$; deshalb ist diese Beobachtung äquivalent dazu, dass die Quotienten $(a_i - a_{i+1})/(a_{i+1} - a_{i+2})$ ungefähr 4 sind.

Dies testen wir mit dem folgenden weiteren Code:

```
print("Quotients of any two subsequent differences of estimates:")
for i in range(m-2):
    q = (results[i] - results[i+1]) / (results[i+1] - results[i+2])
    print(q)
```

Wir erhalten den folgenden Output:

```
Quotients of any two subsequent differences of estimates:
4.164784400584785
4.039182316416593
4.009677144752887
4.002411992937073
4.00060254408483
4.000150607761501
4.000037649528035
4.000009414842847
```

## (3)

Wir berechnen die Approximationen für $\int_0^2 3^{3x-1}\,\mathrm{d}x$ für $m = 1, \ldots, 10$ mit dem folgenden Code:

```
m = 10
f = (lambda x : 3**(3*x-1))
results = powertrapeze( f, 0, 2, m)
print("Calculate trapeze estimate for int. of 3^(3x-1) from 0 to 2, 2^m
    intervals:")
print(" m \testimate")
for i in range(m):
    print("{:2d}\t{:24.20f}".format(i+1, results[i]))
```

Wir erhalten den folgenden Output:

```
Calculate trapeze estimate for int. of 3^(3x-1) from 0 to 2, 2^m intervals:
 m      estimate
 1      130.66666666666665719276
 2       89.58204463929762084717
 3       77.74742639121230070032
 4       74.66669853961546721166
 5       73.88840395800384897029
 6       73.69331521665949935596
 7       73.64451070980437918934
 8       73.63230756098684537392
 9       73.62925664736960129630
10       73.62849391106399821183
```

Da $f$ konvex ist, sind die Approximationen $b_n$ monoton fallend. Die Vermutung lässt sich erneut durch das Betrachten der Quotienten $(b_i - b_{i+1})/(b_{i+1} - b_{i+2})$ überprüfen. Wir nutzen den folgenden Code:

```
print("Quotients of any two subsequent differences of estimates:")
for i in range(m-2):
    q = (results[i] - results[i+1])/(results[i+1] - results[i+2])
    print(q)
```

Wir erhalten den folgenden Output:

```
Quotients of any two subsequent differences of estimates:
3.471562932248868
3.841500716121706
3.958305665211694
3.9894387356667425
3.9973509398114637
3.9993371862347957
3.9998342622879792
3.999958563440565
```

Unsere Vermutung scheint sich zu bestätigen.

## Exercise 3.10

Für die Funktion $f(x) = e^{x^2}$ gilt $f''(x) = (4x^2+2)e^{x^2}$. Da $f''(x) > 0$ auf $[0,1]$ monoton steigend ist, gilt für alle $0 \leq a \leq b \leq 1$, dass

$$|\mathrm{E}(f,a,b)| \leq \frac{(b-a)^3}{12} \max_{a \leq x \leq b} |f''(x)| \leq \frac{(b-a)^3}{12} f''(b)\,.$$

Für alle $n \geq 1$ und $0 \leq k \leq n-1$ gilt deshalb

$$\left| E\left(f, \frac{k}{n}, \frac{k+1}{n}\right)\right| \leq \frac{1}{12n^3}\left(4\left(\frac{k+1}{n}\right)^2 + 2\right) \underbrace{e^{((k+1)/n)^2}}_{\leq e \leq 4} \leq \frac{1}{12n^3}(4+2)\cdot 4 \leq \frac{2}{n^3}\,.$$

Der gesamte Fehler für eine Unterteilung von $[0,1]$ in $n$ Intervalle lässt sich deshalb ingesamt durch

$$n \cdot \frac{2}{n^3} = \frac{2}{n^2}$$

abschätzen. Dabei gilt

$$\frac{2}{n^2} < 10^{-6} \iff n^2 > 2 \cdot 10^6 \iff n > \sqrt{2} \cdot 10^3 \impliedby n > 1500\,.$$

Für das verbesserte Trapezverfahren aus Exercise 3.9 gilt mit $n = 2^m$, dass $n > 1500$ für $m \geq 11$. Wir nutzen nun den folgenden Code, um die entsprechenden Approximationen für $m = 1, \ldots, 11$ zu bestimmen:

```
1 from math import exp
2 f = (lambda x: exp(x**2))
3 m = 11
4 results = powertrapeze(f, 0, 1, m)
5 print("Calculate trapeze estimate for int. of e^(x^2) from 0 to 1, 2^m
      intervals:")
6 for i in range(m):
7     print("m={:2d}\t{:24.20f}".format(i+1, results[i]))
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_10.py
Calculate trapeze estimate for int. of e^(x^2) from 0 to 1, 2^m intervals:
m= 1      1.5715831654586320 8091
m= 2      1.4906788616988553 2865
m= 3      1.4697122764296652 8748
m= 4      1.4644203101494817 0764
m= 5      1.4630941026064285 8148
m= 6      1.4627623485777270 2291
m= 7      1.4626793974185883 2292
m= 8      1.4626586588377739 0621
m= 9      1.4626534741431265 1964
m=10      1.4626521779663752 5538
m=11      1.4626518539219939 2744
```

## Exercise 3.11

Ist $T_n(x) = \sum_{k=0}^{n} x^k/k!$ das $k$-te Taylorpolynom für $f$ an der Entwicklungsstelle 0, so gilt für das Restglied $R_n(x) \coloneqq e^x - T_n(x)$, dass es für jedes $x \in \mathbb{R}$ ein $\xi$ zwschen 0 und $x$ gibt, so dass

$$R_n(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}\xi^n = \frac{e^\xi \xi^n}{(n+1)!}\,.$$

Für alle $x \geq 0$ gilt $e^\xi \leq e^x \leq 3^x$, und somit gilt

$$|R_n(x)| \leq \frac{3^x x^n}{(n+1)!} \qquad \text{für alle } x \geq 0.$$

Für alle $x \leq 0$ gilt $e^\xi \leq e^0 = 1$, und somit

$$|R_n(x)| \geq \frac{(-x)^n}{(n+1)!}.$$

Dies führt zu dem folgenden Code:

```
1  def exp_approx(x):
2      y = 1       # current approx
3      d = 6       # number of digits
4      n = 1
5      fac = 1     # n!
6      if x >= 0:
7          while fac < (3**x) * (x**(n+1)) * 10**d:
8              y += x**n / fac
9              n += 1
10             fac *= n
11     if x < 0:
12         while fac < ((-x)**(n+1)) * 10**d:
13             y += x**n / fac
14             n += 1
15             fac *= n
16     return y
```

Wir testen die Genauigkeit des Programms mit dem folgenden Code:

```
from math import exp
print("Comparison of exp_approx(x) and exp(x) up to 7 digits.")
print("{:>3s}   {:>22s}   {:>22s}   {:13s}".format("x","approximation","exact","difference (10 digits)"))
for x in range(-30,31):
    approx = exp_approx(x)
    exact = exp(x)          # not really exact, but better then the above
    print("{:3d}   {:22.7f}   {:22.7f}   {:13.10f}".format(x, approx, exact, exact-approx))
```

Wir erhalten den folgenden (gekürzten) Output:

```
Comparison of exp_approx(x) and exp(x) up to 7 digits.
  x             approximation                    exact   difference (10 digits)
-30               -0.0000855                0.0000000    0.0000855145
-29                0.0000551                0.0000000   -0.0000550745
-28                0.0000050                0.0000000   -0.0000050079
-27               -0.0000045                0.0000000    0.0000044619
-26               -0.0000014                0.0000000    0.0000013633
-25               -0.0000006                0.0000000    0.0000006464
-24               -0.0000003                0.0000000    0.0000002671
-23               -0.0000000                0.0000000    0.0000000403
-22               -0.0000000                0.0000000    0.0000000071
-21               -0.0000000                0.0000000    0.0000000192
[...]
 21       1318815734.4832141       1318815734.4832146    0.0000004768
 22       3584912846.1315928       3584912846.1315918   -0.0000009537
 23       9744803446.2489052       9744803446.2489033   -0.0000019073
 24      26489122129.8434715      26489122129.8434715    0.0000000000
 25      72004899337.3858795      72004899337.3858795    0.0000000000
 26     195729609428.8387451     195729609428.8387756    0.0000305176
 27     532048240601.7988281     532048240601.7986450   -0.0001831055
 28    1446257064291.4738770    1446257064291.4750977    0.0012207031
 29    3931334297144.0424805    3931334297144.0419922   -0.0004882812
 30   10686474581524.4667969   10686474581524.4628906   -0.0039062500
```

Für etwa $x \geq 23$ und $x \leq -26$ hat unsere Approximation nicht mehr mit der gewünschten Genauigkeit, da die aufzuaddierenden Summanden $x^n/n!$ dann betragsmäßig zu klein werden.

# Exercise 3.12

## (1)

Wir definieren zunächst eine Klasse `TimeOutError`, um ggf. eine passende Fehlermeldung ausgeben zu können.

```
class TimeOutError(Exception):
    pass
```

Wir implementieren das Newton-Verfahren für mit der gewünschten Genauigkeit:

```python
def newton(f, f_prime, x):
    n = 1
    xold = x
    xnew = x
    while n <= 100:
        d = f_prime(xold)
        if d == 0:
            raise ZeroDivisionError("derivative vanishes at {}".format(xold))
        xnew = xold - f(xold)/d
        if 0 <= xnew - xold <= 1.E-7 or 0 <= xold - xnew <= 1.E-7:
            return xnew
        xold = xnew
        n += 1
    raise TimeOutError("the calculation takes too long")
```

## (2)

Wir testen unser Programm anhand der gegebenen Funktion $f(x) = x^2 - 2$ mit dem folgenden Code:

```python
f = (lambda x: x**2 - 2)
fprime = (lambda x: 2*x)
print("Calculating an approximation of sqrt(2):")
print( newton(f, fprime, 1) )
```

Wir erhalten den folgenden Output:

```
$ python exercise_03_12.py
Calculating an approximation of sqrt(2):
1.4142135623730951
```

Dabei stimmen die ersten 15 Nachkommestellen mit dem exakten Ergebnis überein.