

Solutions to Exercises to

**Programming Methods
in Scientific Computing**

David Bauer
Jendrik Stelzner

Letzte Änderung: 22. November 2017

Chapter 3 Python, the Fundamentals

Exercise 3.1

Wir nutzen den folgenden, leicht abgeänderten Code (für besseren Output):

```
1 print("Calculate machine epsilon:")
2 eps = 1.0
3 i = 0
4 while 1+eps > 1:
5     eps = eps/2
6     i += 1
7 eps *= 2
8 print("eps: {}".format(eps))
9 print("iterations: {}".format(i))
10
11 print("With different loop condition:")
12 eps = 1.0
13 i = 0
14 while eps > 0:
15     eps /= 2
16     i += 1
17 eps *= 2
18 print("eps: {}".format(eps))
19 print("iterations: {}".format(i))
```

Wir erhalten den folgenden Output:

```
Calculate machine epsilon:
eps: 2.220446049250313e-16
iterations: 53
With different loop condition:
eps: 0.0
iterations: 1075
```

Der IEEE Standard für Floating-Point-Variablen legt fest, dass bei einer Variable vom Typ double die Mantisse 52 Bit groß ist, und der Exponent 11 Bit. Hierbei ist zu beachten, dass der Exponent in der Darstellung um den Bias von 1023 verschoben wird, falls es sich um eine normalisierte Zahl handelt (Exponent ungleich 0) und um 1022, falls es sich um eine denormalisierte Zahl handelt. Ebenfalls wird der größte Exponent (2047) als **NaN** oder **inf** interpretiert, sodass sich der tatsächliche Exponent der Zahl also im Bereich von -1022 bis $+1023$ befindet.

Das bedeutet, dass die kleinste darstellbare Zahl, die größer als 1 ist, von der Form $(1,00\dots001)_2 = 1 + 2^{-52}$ ist. (Hier steht $(\dots)_2$ für die Binärdarstellung.) Die kleinste darstellbare Zahl, die größer als 0 ist, ist dagegen die Zahl

$$(0,00\dots001)_2 \cdot 2^{-1022} = 2^{-1074}.$$

Der erste Algorithmus rechnet der Reihe nach zunächst die Zahlen 2^{-i} aus und vergleicht dann 1 mit $1 + 2^{-i}$. Bei $i = 52$ ist dies die kleinste Zahl größer als 1, wie oben beschrieben. Im nächsten Durchlauf wird dann `eps` auf 2^{-53} gesetzt. Hier wird nun $1 + \text{eps}$ auf 1 gerundet, die Schleife bricht also ab. Anschließend wird `eps` wieder verdoppelt, sodass sich der folgende Output ergibt:

```
eps: 2.220446049250313e-16
iterations: 53
```

Beim zweiten Algorithmus wird dagegen 2^{-i} mit 0 verglichen. Dies muss bis zum 1074-sten Durchlauf der Schleife nicht gerundet werden. Zu diesem Zeitpunkt hat `eps` den Wert 2^{-1074} , die kleinste positive darstellbare Zahl. Im nächsten Schleifendurchlauf wird `eps` halbiert und auf 0 gerundet. Die Schleife bricht also nach 1075 Durchläufen ab. Danach wird `eps` wieder verdoppelt, hat also den Wert $0 \cdot 2 = 0$. Der Output ist also:

```
eps: 0.0
iterations: 1075
```

Exercise 3.2

Wir erweitern die gegebene 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):
37         return self.coeff == other.coeff
38
39     def derivative(self):
40         coeff = []
41         for i in range(1, len(self.coeff)):
42             coeff.append(i * self.coeff[i])
43         return Polynomial(coeff)
44
45     def antiderivative(self):
46         coeff = [0]
47         for i in range(len(self.coeff)):
48             coeff.append(self.coeff[i]/(i+1))
49         return Polynomial(coeff)

```

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

```

1 p = Polynomial([1,2,3])
2 print("The given polynomial p:")
3 print(p)
4 print("The derivative of p:")
5 print( p.derivative() )
6 print("The antiderivative of p:")
7 print( p.antiderivative() )
8 print("Taking antiderivative and then derivative:")
9 print( p.antiderivative().derivative() )

```

Dabei erhalten wir den folgenden Output:

```

$ python exercise_03_02.py
The given polynomial 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.

```

1 class Matrix():
2     def __init__(self, entries):
3         m = len(entries)
4         if m == 0:
5             raise ValueError("height must be positive")
6         n = len(entries[0])
7         if n == 0:
8             raise ValueError("width must be positive")
9         for i in range(1, m):
10            if len(entries[i]) != n:
11                raise ValueError("rows must have the same width")
12        self.height = m
13        self.width = n
14        self.entries = entries
15
16    def __getitem__(self, i):      # allows to get the rows via A[i]
17        return self.entries[i]
18
19    def __setitem__(self, i, k):   # allows to set rows via A[i]
20        self.entries[i] = k
21
22    def __str__(self):            # allows print(A) for a Matrix A
23        rows = ["["]*self.height
24        for j in range(self.width): # construct output columnwise, align left
25            numbers = []           # numbers to appear in column j
26            maxlen = 0             # maximal length of a number in column j
27            for i in range(self.height):
28                s = str(self[i][j])
29                numbers.append(s)
30                if len(s) > maxlen:
31                    maxlen = len(s)
32            for i in range(self.height):
33                # pad the entries if they are too short
34                rows[i] += numbers[i] + " "*(maxlen-len(numbers[i])) + " "
35        s = ""
36        for r in rows:
37            s += r[:-1] + "]\n" # remove white space at the end of ech line
38        s = s[:-1]             # remove empty line at the end
39        return s
40
41    def __repr__(self):
42        return str(self)
43
44    def __mul__(self, other):
45        if self.width != other.height:
46            raise TypeError('matrix dimensions do not match')
47        newentries = []
48        for i in range(self.height):
49            row = []
50            for j in range(other.width):
51                s = self[i][0] * other[0][j] # makes s have the right type
52                for k in range(1, self.width):
53                    s += self[i][k] * other[k][j]
54            row.append(s)
55        newentries.append(row)

```

```

56         return Matrix(newentries)
57
58     def __eq__(self, other):
59         if self.height != other.height or self.width != other.width:
60             return False
61         for i in range(self.height):
62             for j in range(self.width):
63                 if self[i][j] != other[i][j]:
64                     return False
65         return True
66
67     def mapentries(self, f):          # applies a function to all entries
68         A = zeromatrix(self.height, self.width) # zeromatrix is defined below
69         for i in range(self.height):
70             for j in range(self.width):
71                 A[i][j] = f(self[i][j])
72         return A
73
74     def addrow(self, i, j, c):        # add c times row j to row i
75         for k in range(self.width):
76             self[i][k] = c * self[j][k] + self[i][k]
77             # makes c responsible for implementing the operations
78
79     def addcolumn(self, i, j, c):    # add c times column j to column i
80         for k in range(self.height):
81             self[k][i] = c * self[k][j] + self[k][i]
82
83     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

```

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 denominator 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("denominator 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,
28                             self.denum * other.denum )
29         raise TypeError("unsupported operand type(s) for + or add(): '
30             Rational' and '{}'.format(type(other).__name__)
31
32     def __sub__(self, other):
33         if type(other) == int:
34             return self - Rational(other)
35         elif type(other) == Rational:
36             return Rational( self.num * other.denum - self.denum * other.num,
37                             self.denum * other.denum )
38         raise TypeError("unsupported operand type(s) for - or sub(): '
39             Rational' and '{}'.format(type(other).__name__)
40
41     def __mul__(self, other):
42         if type(other) == int:
43             return self * Rational(other)
44         elif type(other) == Rational:
45             return Rational( self.num * other.num, self.denum * other.denum )
46         raise TypeError("unsupported operand type(s) for * or mul(): '
47             Rational' and '{}'.format(type(other).__name__)
48
49     def __truediv__(self, other):
50         if type(other) == int:
51             return self / Rational(other)
52         elif type(other) == Rational:
53             if other.num == 0:
54                 raise ZeroDivisionError("division by zero")
55             return Rational( self.num * other.denum, self.denum * other.num)
56         raise TypeError("unsupported operand type(s) for / or truediv(): '
57             Rational' and '{}'.format(type(other).__name__)
58
59     def __pow__(self, n):    # supports only integer powers
60         if not type(n) == int:
61             raise TypeError("unsupported operand type(s) for ** or pow(): '
62                 Rational' and '{}'.format(type(n).__name__)
63         if n >= 0:
64             return Rational(self.num**n, self.denum**n)
65         return Rational(self.denum, self.num)**(-n)
66
67     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:

```

1  from copy import deepcopy
2
3  # expects the matrix entries to be comparable to 0 in a sensible way
4  def naive_lu(A):
5      if A.height != A.width:
6          raise ValueError("matrix is not square")
7      U = deepcopy(A) # circumvent pass by reference
8      n = U.height
9      L = identitymatrix(n)
10     # bring U in upper triangular form, change L such that always LU = A
11     for j in range(n):
12         if U[j][j] == 0:
13             raise ValueError("algorithm does not work for this matrix")
14         for i in range(j+1,n):
15             L.addcolumn(j, i, U[i][j]/U[j][j]) # important: change L first
16             U.addrow(i, j, -U[i][j]/U[j][j])
17     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:

```

1  A = Matrix([[3,2,1],[6,6,3],[9,10,6]])
2  print("A:")
3  print(A)
4  (L,U) = naive_lu(A.mapentries(Rational))
5  print("L:")
6  print(L)

```

```

7 | print("U:")
8 | print(U)
9 | B = Matrix([[0,1],[1,0]])
10 | print("B:")
11 | print(B)
12 | print("Trying to calculate the LU decomposition of B:")
13 | (L,U) == naive_lu(B)

```

Als Output erhalten wir das Folgende:

```

$ 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 copy import deepcopy
2 | from math import sqrt
3 |
4 | # expects int or float as matrix entries
5 | def cholesky(A):
6 |     if A.height != A.width:
7 |         raise ValueError("matrix is not square")
8 |     B = deepcopy(A)
9 |     n = B.height
10 |    L = zeromatrix(n,n)
11 |    for i in range(n):

```

```

12         rowsum = 0
13         for j in range(i):
14             s = 0
15             for k in range(j):
16                 s += L[i][k] * L[j][k]
17             L[i][j] = (B[i][j] - s)/L[j][j]
18             rowsum += L[i][j]**2
19         L[i][i] = sqrt( B[i][i] - rowsum )
20     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())
9 B = Matrix([[1.01E-2, 0.705, 1.42E-2],[0.705,49.5,1],[1.42E-2,1,1]])
10 print("B:")
11 print(B)
12 L = cholesky(B)
13 print("L:")
14 print(L)
15 print("L * L^T:")
16 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.

```
1 from copy import deepcopy
2
3 # expects the matrix entries to be comparable to 0 in a sensible way
4 def invert(A):
5     if A.height != A.width:
6         raise ValueError("matrix is not square")
7     B = deepcopy(A) # circumvent pass by reference
8     n = B.height
9     Inv = identitymatrix(n)
10    B = B.mapentries(Rational) # make all
11    Inv = Inv.mapentries(Rational) # entries rational
12    # bring B in lower triangular form
13    for j in range(n):
14        p = -1
15        for i in range(j,n):
16            if B[i][j] != 0:
17                p = i
18                break
19        if p == -1:
20            raise ZeroDivisionError("matrix is not invertible")
21        for i in range(p+1,n):
22            Inv.addrow(i, p, -B[i][j]/B[p][j]) # import: change inverse
23            # first
24            B.addrow(i, p, -B[i][j]/B[p][j])
25        # norm the diagonal entries
26        for i in range(n):
27            Inv.multrow(i, B[i][i]**(-1)) # **(-1) also works for Rational
28            B.multrow(i, B[i][i]**(-1))
29        # bring B into identity form
30        for j in range(n):
31            for i in range(j):
32                Inv.addrow(i, j, -B[i][j])
33                B.addrow(i, j, -B[i][j])
34    return Inv
```

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:

```

1 from copy import deepcopy
2 from math import sqrt
3
4 # assumes the matrix to have integer or float values
5 # and to be nonsingular
6 def qrdecomp(A):
7     if A.height != A.width:
8         return ValueError("only square matrices are supported")
9     n = A.height
10    Q = deepcopy(A)
11    R = identitymatrix(n)
12    for j in range(n):
13        # make the j-th column of Q orthogonal to the next columns
14        for k in range(j):
15            s = 0 # inner product of j-th and k-th columns
16            for i in range(n):
17                s += Q[i][j] * Q[i][k]
18            Q.addcolumn(j, k, -s)

```

```

19         R.addrow(k, j, s)
20         sn = 0 # squared norm of the j-th column
21         for i in range(n):
22             sn += Q[i][j]**2
23         Q.multcolumn(j, sqrt(sn)**(-1))
24         R.multrow(j, sqrt(sn))
25     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:

```

1 A = Matrix([[12,-51,4],[6,167,-68],[-4,24,-41]])
2 print("A:")
3 print(A)
4 (Q,R) = qrdecomp(A)
5 print("Q:")
6 print(Q)
7 print("Q * Q^T:")
8 print(Q * Q.transpose())
9 print("R:")
10 print(R)
11 print("Q*R:")
12 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 das mehrfache Berechnen gleicher Funktionswerte 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 $\int_0^\pi \sin(x) dx$ 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:

```
1 def powertrapeze(f, a, b, mmax):
2     integrals = []          # list of the approximations
3     values = [f(a), f(b)]   # list of the calculated values
4     for m in range(1,mmax+1):
5         n = 2**m
6         for k in range(1,n,2):
7             values.insert(k, f(a + (k/n)*(b-a)))  # add new values
8     integral = 0
```

```

9         for i in range(len(values)-1):
10             integral += values[i] + values[i+1]
11             integral = (b-a)*integral/n/2
12             integrals.append(integral)                # add new approx.
13     return integrals

```

Hiermit berechnen die Approximationen für $\int_0^\pi \sin(x) dx$ für $m = 1, \dots, 10$ mit dem folgenden Code:

```

1 from math import sin, pi
2 m = 10
3 results = powertrapeze(sin, 0, pi, m)
4 print("Calculate trapeze estimate for int. of sin from 0 to pi, 2^m intervals
   :")
5 print(" m \testimate \t\terror")
6 for i in range(m):
7     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 $a_0 \leq a_1 \leq \dots \leq a_n$, da \sin auf $[0, \pi]$ konkav ist. Deshalb ist die Vermutung 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:

```

1 print("Quotients of any two subsequent differences of estimates:")
2 for i in range(m-2):
3     q = (results[i] - results[i+1]) / (results[i+1] - results[i+2])
4     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
```

Es fällt auf, dass das Verhältnis sogar gegen 4 zu gehen scheint.

(3)

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

```
1 m = 10
2 f = (lambda x : 3**(3*x-1))
3 results = powertrapeze( f, 0, 2, m)
4 print("Calculate trapeze estimate for int. of 3^(3x-1) from 0 to 2, 2^m
   intervals:")
5 print(" m \testimate")
6 for i in range(m):
7     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. Hierfür nutzen wir (erneut) den folgenden Code:

```
1 print("Quotients of any two subsequent differences of estimates:")
2 for i in range(m-2):
3     q = (results[i] - results[i+1])/(results[i+1] - results[i+2])
4     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

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

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 bei einer Unterteilung von $[0, 1]$ in n Intervalle lässt sich deshalb durch

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

abschätzen. Dabei gilt

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

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 $m = 1, \dots, 11$ 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      1.57158316545863208091
m= 2      1.49067886169885532865
m= 3      1.46971227642966528748
m= 4      1.46442031014948170764
m= 5      1.46309410260642858148
m= 6      1.46276234857772702291
m= 7      1.46267939741858832292
m= 8      1.46265865883777390621
m= 9      1.46265347414312651964
m=10     1.46265217796637525538
m=11     1.46265185392199392744

```

Exercise 3.11

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

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

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

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

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:

```

1 from math import exp
2 print("Comparison of exp_approx(x) and exp(x) up to 7 digits.")
3 print("{:>3s}   {:>22s}   {:>22s}   {:>13s}".format("x", "approximation", "exact",
4     "difference (10 digits)"))
5 for x in range(-30,31):
6     approx = exp_approx(x)
7     exact = exp(x)          # not really exact, but better than the above
8     print("{:3d}   {:22.7f}   {:22.7f}   {:>13.10f}".format(x, approx, exact,
9         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			
-29			
-28			
-27			
-26			
-25			
-24			
-23			
-22			
-21			
-20			
-19			
-18			
-17			
-16			
-15			
-14			
-13			
-12			
-11			
-10			
-9			
-8			
-7			
-6			
-5			
-4			
-3			
-2			
-1			
0	1	1	0
1	2.71828	2.71828	0
2	7.38906	7.38906	0
3	20.0855	20.0855	0
4	54.5981	54.5981	0
5	148.413	148.413	0
6	403.429	403.429	0
7	1096.63	1096.63	0
8	2980.96	2980.96	0
9	8103.08	8103.08	0
10	22026.5	22026.5	0
11	59874.1	59874.1	0
12	163754	163754	0
13	442413	442413	0
14	1184150	1184150	0
15	3154840	3154840	0
16	8387910	8387910	0
17	22140600	22140600	0
18	58347600	58347600	0
19	154990000	154990000	0
20	405220000	405220000	0
21	1064150000	1064150000	0
22	2754140000	2754140000	0
23	7136560000	7136560000	0
24	18500000000	18500000000	0
25	47525000000	47525000000	0
26	122145000000	122145000000	0
27	315484000000	315484000000	0
28	810308000000	810308000000	0
29	2094830000000	2094830000000	0
30	5459810000000	5459810000000	0

-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 die gewünschten Genauigkeit, da die aufzuaddierenden Summanden $x^n/n!$ dann zu klein werden.

Exercise 3.12

(1)

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

```
1 class TimeoutError(Exception):
2     pass
```

Wir implementieren das Newton-Verfahren mit der gewünschten Genauigkeit:

```
1 def newton(f, f_prime, x):
2     n = 1
3     xold = x
4     xnew = x
5     while n <= 100:
6         d = f_prime(xold)
7         if d == 0:
8             raise ZeroDivisionError("derivative vanishes at {}".format(xold))
9         xnew = xold - f(xold)/d
10        if 0 <= xnew - xold <= 1.E-7 or 0 <= xold - xnew <= 1.E-7:
11            return xnew
12        xold = xnew
13        n += 1
14    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:

```
1 f = (lambda x: x**2 - 2)
2 fprime = (lambda x: 2*x)
3 print("Calculating an approximation of sqrt(2):")
4 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.