Solutions to Exercises to

Programming Methods in Scientific Computing

David Bauer
Jendrik Stelzner

Letzte Änderung: November 16, 2017

Chapter 3: Python, the Fundamentals

Exercise 3.2

We extend the given class `Polynomial` by functions for the derivative and antiderivative:

```
class Polynomial:
    def __init__(self, coefficients):
        self.coeff = coefficients

    def __call__(self, x):
        s = 0
        for i in range(len(self.coeff)):
            s += self.coeff[i]*x**i
        return s

    def __add__(self, other):
        l = []
        if len(self.coeff) > len(other.coeff):
            l += self.coeff
            for i in range(len(other.coeff)):
                l[i] += other.coeff[i]
        else:
            l += other.coeff
            for i in range(len(self.coeff)):
                l[i] += self.coeff[i]
        return Polynomial(l)

    def __eq__(self, other):
        return self.coeff == other.coeff

    def derivative(self):
        coeff = []
        for i in range(1,len(self.coeff)):
            coeff.append(i * self.coeff[i])
        return Polynomial(coeff)

    def antiderivative(self):
        coeff = [0]
        for i in range(len(self.coeff)):
            coeff.append(self.coeff[i]/(i+1))
        return Polynomial(coeff)
```

For the given polynomial $p(x) = 3x^2 + 2x + 1$ we get the following results:

```
    >>> p = Polynomial([1,2,3])
    >>> p.derivative().coeff
    [2, 6]
    >>> p.antiderivative().coeff
```

```
    [0, 1.0, 1.0, 1.0]
    >>> p.antiderivative().derivative().coeff
    [1.0, 2.0, 3.0]
```

Exercise 3.4

```
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):
        rows = ["["]*self.height
        for j in range(self.width): # build the output columnwise
            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 __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]    # s has 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
```

For the matrices

```
    >>> A = matrix([[0,1],[1,0],[1,1]])
    >>> B = matrix([[1,2,3,4],[5,6,7,8]])
    >>> C = matrix([[1,0],[0,1],[1,0],[0,1]])
    >>> A * (B * C) == (A * B) * C
    True
```

Exercise 3.5

We first implement a class `Rational` to allow calculations with rational numbers with arbitrary precision.

```
class Rational():
    def __init__(self, num, denum = 1): # default denumerator is 1
        if type(num) != int:
            raise TypeError("numerator is no integer")
        if type(denum) != int:
            raise TypeError("denumerater is no integer")
        if denum == 0:
            raise ZeroDivisionError("denumerator is zero")
        self.num = num
        self.denum = denum

    def __str__(self):  # allows print(A) for a matrix A
        return "{}/{}".format(self.num, self.denum)

    def __add__(self, other):
        return Rational( self.num * other.denum + self.denum * other.num, self.denum * other.denum )

    def __sub__(self, other):
        Rational( self.num * other.denum - self.denum * other.num, self.denum * other.num )

    def __neg__(self):
        return Rational( -self.num, self.denum )

```

```
    def __mul__(self, other):
        return Rational( self.num * other.num, self.denum * other.denum )

    def __truediv__(self, other):
        if other.num == 0:
            raise ZeroDivisionError("division by zero")
        return Rational( self.num * other.denum, self.denum * other.num)

    def inverse(self):  # short hand notation
        return Rational(1)/self

    def __eq__(self, other):
        if type(other) == int:  # allows comparison to int, used for x == 0
            return (self == Rational(other))
        return (self.num * other.denum == self.denum * other.num)

    def __float__(self):
        return self.num / self.denum
```

Next we extend the previous `matrix` class.

```
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):
        rows = ["["]*self.height
        for j in range(self.width): # build the output columnwise
            numbers = []            # the numbers to appear in column j
            maxlen = 0              # the maximal length of a number 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 each line
        s = s[:-1]              # remove an empty line
        return s

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

    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 map(self, f):   # applies a function to all entries
        for i in range(self.height):
            for j in range(self.width):
                self[i][j] = f(self[i][j])

    def swaprows(self, i, j):       # swap rows i and j
        if i > self.height or j > self.height:
            raise ValueError("swap nonexistent rows")
        l = self[i]
        self[i] = self[j]
        self[j] = l

    def addrow(self, i, j, c=1):    # add c times row j to row i
        for k in range(self.width):
            self[i][k] = self[i][k] + self[j][k] * c

    def addcolumn(self, i, j, c=1): # add c times column j from colunm i
        for k in range(self.height):
            self[k][i] = self[k][i] + self[k][j] * c

    def multrow(self, i, c):        # multiply row i with c
        for j in range(self.width):
            self[i][j] *= c
```

We also define some auxiliary matrix functions:

```
def zeromatrix(height, width):  # creates a zero matrix
    entries = []
    for i in range(height):
        entries.append([0]*width)
```

```
    return matrix(entries)

def identitymatrix(size):   # creates an identiy matrix
    E = zeromatrix(size, size)
    for i in range(size):
        E[i][i] = 1
    return E

def copymatrix(A):  # copies a matrix
    B = zeromatrix(A.height, A.width)
    for i in range(A.height):
        for j in range(A.width):
            B[i][j] = A[i][j]
    return B
```

The LU decomposition of some matrices can be computed by the following naive algorithm:

```
def naive_lu(A):
    if A.height != A.width:
        raise ValueError("matrix is not square")
    U = copymatrix(A) # circumvent pass by reference
    n = U.height
    L = identitymatrix(n)
    U.map(Rational)   # make all
    L.map(Rational)   # entries rational
    # bring U in upper triangular form, change L such that 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)
```

For the given matrix

$$A = \begin{pmatrix} 3 & 2 & 1 \\ 6 & 6 & 3 \\ 9 & 10 & 6 \end{pmatrix}$$

we get the following result:

```
>>> A = matrix([[3,2,1],[6,6,3],[9,10,6]])
>>> (L,U) = naive_lu(A)
>>> print(L)
[9/9  0/18  0/1]
[18/9 18/18 0/1]
[27/9 36/18 1/1]
>>> print(U)
[3/1   2/1   1/1    ]
[0/3   6/3   3/3    ]
[0/162 0/162 162/162]
>>> B = L*U
>>> B.map(float)
>>> print(B)
[3.0 2.0  1.0]
```

```
[6.0 6.0  3.0]
[9.0 10.0 6.0]
```

For the given matrix

$$B = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

(which has no LU decomposition) the program stops with an error:

```
>>> B = matrix([[0,1],[1,0]])
>>> (L,U) == naive_lu(B)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 12, in naive_lu
ValueError: algorithm does not work for this matrix
```

Exercise 3.7

We use the classes `Rational` and `matrix` as in Exercise 3.5, as well as the auxiliary functions. We can then use the Gauß algorithm:

```
# allowed input are integer matrices
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.map(Rational)     # make all
    Inv.map(Rational)   # entries rational
    # bring B in lower triangular form
    for j in range(n):
        p = -1
        for i in range(j,n):
            if B[i][j] != 0:
                p = i
                break
        if p == -1:
            raise ZeroDivisionError("matrix is not invertible")
        for i in range(p+1,n):
            Inv.addrow(i, p, -B[i][j]/B[p][j])  # import: change inverse first
            B.addrow(i, p, -B[i][j]/B[p][j])
    # norm the diagonal entries
    for i in range(n):
        Inv.multrow(i, B[i][i].inverse())
        B.multrow(i, B[i][i].inverse())
    # bring B into identity form
    for j in range(n):
        for i in range(j):
            Inv.addrow(i, j, -B[i][j])
            B.addrow(i, j, -B[i][j])
    return Inv
```

For the given matrix

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

this results in the following output:

```
>>> A = matrix([[3,-1,2],[-3,4,-1],[-6,5,-2]])
>>> B = invert(A)
>>> A.map(Rational)
>>> A*B == identitymatrix(3)
True
>>> B.map(float)
>>> print(B)
[-0.3333333333333333 0.8888888888888888 -0.7777777777777778]
[0.0                 0.6666666666666666 -0.3333333333333333]
[1.0                 -1.0               1.0                ]
```