大作业技术报告

*额外补充的函数：高斯函数

```
def gauss(self):
    a = self.copy().data
    i, p = 0, 0

    while i < min(self.dim[0], self.dim[1]):
        while p < len(a[0]) and a[i][p] == 0:
            for j in range(i + 1, self.dim[0]):
                if a[j][p] != 0:
                    a[i], a[j] = a[j], a[i]
                    break
            if a[i][p] == 0:
                p += 1

        for k in range(self.dim[0]):
            if k != i and p < len(a[0]):
                d = a[k][p]
                for j in range(len(a[0])):
                    a[k][j] -= a[i][j] * d / a[i][p]   # 这里加了一个/a[i][p]

        i += 1
        p += 1

    return Matrix(a)
```

（一）功能说明

1. 首先，通过 a = self.copy().data 复制并获取矩阵数据，初始化索引 i 和 p 开启循环处理。外层 while 循环以行数和列数较小值为界限按行遍历矩阵。内层循环在当前列遇到元素为 0 时，会从下一行开始查找非零元素行并交换，确保主元非零。
2. 接着，关键的消元操作部分，for k in range(self.dim[0])循环遍历所有行，对于非当前行（k!= i）且列索引在合理范围时，通过计算系数 d，按照高斯消元思路，在列维度上对元素做相应减法操作，逐步将矩阵化为行阶梯形式。

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(A.gauss())
```

结果是：

```
[[1.0 0.0 -1.0]
 [0.0 -3.0 -6.0]]
```

第一个函数：初始化

```
def __init__(self, data=None, dim=None, init_value=0):
    self.data = data
    self.dim = dim
    self.init_value = init_value
    if self.dim == None and self.data == None:
        return None
    elif self.data == None:
        self.data = [[self.init_value for _ in range(dim[1])] for _ in range(dim[0])]
    elif self.dim == None:
        if type(self.data[0]) != int:   # 防止生成一维矩阵时报错
            self.dim = (len(self.data), len(self.data[0]))
        else:
            self.dim = (len(self.data), self.data[0])
    elif self.dim != (len(self.data), len(self.data[0])):
        if type(self.data[0]) != int:
            self.dim = (len(self.data), len(self.data[0]))
        else:
            self.dim = (len(self.data), self.data[0])
```

（一）功能说明

__init__()对对象进行初始化

先将输入的内容赋值给变元方便后续调用

再判断 dim 和 data 是否为 None

1） 如果二者均为 None 那么返回 None，print 时输出 Error!!!

2） 1）不成立的条件下，如果 data 为 None（即 data 为 None 但是 dim 不为 None，则根据 dim 创建元素全为 init_value 的矩阵

3） 1）2）均不成立的条件下，如果 dim 为 None（即 data 不为 None，dim 为 None）根据 data 的 dim 通过测量 data 含有几个列表（即几行）和第一行的列表含有几个元素（即几列）设置对象的 dim

Debug 时发现如果为一维矩阵，无法直接获得 len(self.data[0])，所以加了判断条件，单独设置一维矩阵的 data

4） 在 1，2，3 均不成立，且 dim 与 data 不相等的条件下（即 data 与 dim 均不为 0 但是不相等），将 data 的 dim 赋值给 dim，同 3 进行了分类讨论

（二）调试

```
import minimatrix1 as mm
1. 第一种情况:
mat1 = mm.Matrix(dim=(2, 3), init_value=0)
print(mat1)
```

结果是:

```
[[0 0 0]
```

[0 0 0]]
2. 第二种情况：
mat2 = mm.Matrix(data=[[0, 1], [1, 2], [2, 3]])
print(mat2)
结果是：
[[0 1]
[1 2]
[2 3]]

第二个函数：矩阵形状

```
def shape(self):
    if self.data != None:
        return self.dim[0], self.dim[1]
    elif self.dim != None:
        return self.dim
    else:
        raise MatrixError("请输入参数！")
```

（一）功能说明
我们用两位数字分别表示矩阵的行数和列数，进而表示矩阵的形状。
1. 如果矩阵的数据存在，则矩阵的形状直接就是这组矩阵数据对应的元组中的两位数字。用 dim[0]和 dim[1]分别获取元组中的两个数字，就是矩阵的形状。
2. Dim 元组中的两个数字直接表示矩阵的行数和列数，如果 dim 中的数据存在，则可直接返回这两个数字，就是矩阵的形状。
3. 如果矩阵的数据与 Dim 元组中的两个数字均不存在，则该矩阵未知，不可得出其形状。所以会返回“请输入参数！”，提示操作者给出相关的数据方便确定矩阵的形状。

（二）调试
import minimatrix1 as mm
1. 第一种情况，已知矩阵的数据。
   a = mm.Matrix(data=[[1, 2, 3], [4, 5, 6], [7, 8, 9]])
   print(a.shape())
   结果是：（3，3）
2. 第二种情况，已知矩阵的 dim。
   a = mm.Matrix(dim=(3, 3))
   print(a.shape())
   结果是：（3，3）
3. 第三种情况，矩阵的数据与 Dim 元组中的两个数字均不存在。
   a = mm.Matrix()
   print(a.shape())
   结果是：Error!!!

第三个函数：改变矩阵形状

```
def reshape(self, newdim):
    lst1 = []
lst2 = [[0 for _ in range(newdim[1])] for _ in range(newdim[0])]

if (newdim[0] * newdim[1] != self.dim[0] * self.dim[1]):
    raise MatrixError("矩阵维数不匹配！")
else:
    for i in range(self.dim[0]):
        for j in range(self.dim[1]):
            lst1.append(self.data[i][j])

    k = 0
    for k in range(newdim[0] * newdim[1]):
        lst2[k // newdim[1]][k % newdim[1]] = lst1[k]

    return Matrix(lst2)
```

（一）功能说明

1.先创建一个和 self 的行数与列数乘积相同的 lst。
2.判定原矩阵和新矩阵的维数是否相同，若不相同，直接报错。
3.若相同，则将 self 的元素摊开到新创建的 lst 中。。
4.最后按照目标格式将一整行 lst 截断，匀到新的 lst 中

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(A.reshape((3, 2)))
```

结果是：

```
[[1 2]
 [3 4]
 [5 6]]
```

第四个函数：矩阵乘法

```
def dot(self, other):
    if len(self.data[0]) != len(other.data):
        raise MatrixError("矩阵维数不匹配，无法相乘！")
    else:
        result_data = [[0 for _ in range(len(other.data[0]))] for _ in range(len(self.data))]
        for i in range(len(self.data)):
            for j in range(len(other.data[0])):
```

```
                for k in range(len(other.data)):
                    result_data[i][j] += self.data[i][k] * other.data[k][j]
        return Matrix(data=result_data)
```

（一）功能说明

1. 判断两个矩阵能否相乘：若第一个矩阵的列数等于第二个矩阵的行数，则可以进行矩阵乘法。否则就会返回“矩阵维数不匹配，无法相乘！”。
2. 定义嵌套列表 result_data：相乘后的矩阵的行数等于第一个矩阵的行数，列数等于第二个矩阵的列数。所以初始化一个嵌套列表 result_data，代表相乘后的矩阵，且矩阵中的每一个元素的初始值都是 0。
3. 乘法运算：让嵌套列表 result_data（i，j）位置处的元素的数值等于第一个列表第 i 行的所有元素与第二个列表第 j 列所有元素的乘积之和。所以用 k 去遍历所有的乘积，最后进行求和。

（二）调试

```
import minimatrix1 as mm
1.第一种情况，可以进行矩阵乘法。
a = mm.Matrix(data = [[1,2,3],[6,5,4]])
b = mm.Matrix(data = [[1,2],[3, 4], [5, 6]])
print(a.dot(b))
结果为：[[22 28]
         [41 56]]
2. 第二种情况，不可以进行矩阵乘法。
a = mm.Matrix(data = [[1,2,3],[6,5,4]])
b = mm.Matrix(data = [[1,2],[3, 4]])
print(a.dot(b))
结果为：Error!!!
```

第五个函数：矩阵转置

```
def T(self):
    a = [[0 for _ in range(self.dim[0])] for _ in range(self.dim[1])]
    for i in range(self.dim[0]):
        for j in range(self.dim[1]):
            a[j][i] = self.data[i][j]
    self.data = a
    self.dim = (self.dim[1], self.dim[0])
```

（一）功能说明

1. 先设置一个与转置后的矩阵 dim 相同的全 0 矩阵，通过循环将原矩阵中第 i 行第 j 列的元素的值赋值给转置后第 j 行第 i 列的元素
2. 重新设置矩阵的 data 和 dim

（二）调试

```
import minimatrix1 as mm
B = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
B.T()
print(B)
```
结果是：
```
[[1 4]
 [2 5]
 [3 6]]
```

第六个函数：对应元素加法

```
def sum(self, axis=None):
    if axis == 0:
        sum1 = [[0 for _ in range(self.dim[1])]]
        for i in range(self.dim[1]):
            for j in range(self.dim[0]):
                sum1[0][i] += self.data[j][i]
        return Matrix(sum1)

    elif axis == 1:
        self.T()
        max1 = self.sum(0)
        self.T()
        max1.T()
        return max1

    else:
        max2 = self.sum(1)
        max3 = max2.sum(0)
        return max3
```

（一）功能说明

1. 若把同一列的元素数值加到一行，那么依次读取每一行中指定列的元素并相加
2. 将加和的元素添加到新 lst 中，整合输出
3. 若把同一行的元素数值加到一列，那么先对原矩阵作转置，进行上述求和
4. 然后再将得到的行矩阵进行转置，得到目标列矩阵
5. 若加和所有元素，则相当于先对行求和，再对列求和，得到一个单独的数，组成新矩阵

（二）调试

```
import minimatrix1 as mm
```

1. 第一种情况：

   ```
   A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
   print(A.sum())
   ```

   结果是：[[21]]

2. 第二种情况：
   A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
   print(A.sum(1))
   结果是：
   [[6]
    [15]]
3. 第三种情况：
   A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
   print(A.sum(0))
   结果是：[[5 7 9]]

第七个函数：矩阵备份

做法一：deepcopy

```
def copy(self):
    a = copy.deepcopy(self.data)
    return Matrix(a)
```

（一）功能说明

使用了 Python 标准库中的 copy 模块里的 deepcopy 函数，对 self.data 进行深拷贝操作，获取到一个和自身数据内容相同，但在内存中是独立存在的另一个 Matrix 实例。

（二）调试

```
import minimatrix1 as mm
a = mm.Matrix(data = [[1,2,3],[6,5,4]])
print(a.copy())
```

结果为：[[1 2 3]
[6 5 4]]

做法二：

```
def copy(self):
    lst = self.data
    lst1 = [[0 for _ in range(self.dim[1])] for _ in range(self.dim[0])]
    for i in range(len(lst)):
        for j in range(len(lst[0])):
            lst1[i][j] = lst[i][j]
    return Matrix(lst1)
```

（一）功能说明

1. 用 lst 存储已知矩阵的数据。
2. 新建 lst1：lst1 的形状与 lst 相同，但是每个位置的数据都是 1。
3. 遍历：通过遍历 lst 每个位置上的元素，并给 lst1 相应位置上的元素赋相同的值，使 lst1 与 lst 的数据完全相同，是同一个矩阵。

解释：使 lst1 与 lst 的各个位置上的数据完全相同，代表同一个矩阵。但是由于创建的时候

是不同的嵌套列表，所以它们在计算机中存储的位置不同，即它们的 id 不同。这就实现的矩阵的备份。

（二）调试

```
import minimatrix1 as mm
a = mm.Matrix(data = [[1,2,3],[6,5,4]])
print(a.copy())
```

结果为：[[1 2 3]
[6 5 4]]

第八个函数：矩阵 Kronecker 积

```
def Kronecker_product(self, other):
    a = Matrix(None, (self.dim[0] * other.dim[0], self.dim[1] * other.dim[1]), 0)
    for i in range(a.dim[0]):
        for j in range(a.dim[1]):
            b = i // other.dim[0]
            c = j // other.dim[1]
            d = i % other.dim[0]
            e = j % other.dim[1]
            a.data[i][j] = self.data[b][c] * other.data[d][e]
    return a
```

（一）功能说明

1. 先设置一个和结果的 Matrix 对象 dim 一样的全 0 矩阵
2. 根据定义，行数等于 self 的行数乘上 other 的行数，列数等于 self 的列数乘上 other 的行数；要找到用哪个 self 的元素乘，就是要找到这是复制的第几个 other，所以去找行号和列号除以 other 的行数和列数的商；要找到用哪个 other 的元素乘，就是要找到这是复制的某个 other 中的第几行和第几列，所以是取余数

Debug 时一直用的是 self 的 dim 作为除数，所以一直不对，后来重新梳理了概念得到了正确结果

（二）调试

```
import minimatrix1 as mm
ma1 = mm.Matrix([[1,2,3],[6,5,4]],(2,3))
ma2 = mm.Matrix([[1,4,2],[1,1,2],[1,2,1]],(3,3))
print(ma1.Kronecker_product(ma2))
```

结果是：

```
[[1 4 2 2 8 4 3 12 6]
 [1 1 2 2 2 4 3 3 6]
 [1 2 1 2 4 2 3 6 3]
 [6 24 12 5 20 10 4 16 8]
 [6 6 12 5 5 10 4 4 8]
 [6 12 6 5 10 5 4 8 4]]
```

第九个函数：矩阵索引

```
def __getitem__(self, key):
    if type(key) == tuple and len(key) == 2 \
            and type(key[0]) == int and type(key[1]) == int:
        return self.data[key[0]][key[1]]
    else:
        x1, x2 = key
        sa1, so1 = x1.start, x1.stop
        if sa1 == None:
            sa1 = 0
        if so1 == None:
            so1 = self.dim[0]
        x1 = slice(sa1, so1)

        sa2, so2 = x2.start, x2.stop
        if sa2 == None:
            sa2 = 0
        if so2 == None:
            so2 = self.dim[1]
        x2 = slice(sa2, so2)

        return Matrix([[self.data[i][j] for j in range(x2.start, x2.stop)] \
                       for i in range(x1.start, x1.stop)])
```

（一）功能说明

1. 先判断输入的 key 的格式，若发现是形如(1,2)的单个元素指定 key，则读取 self 中该行该列的元素
2. 若不是这种情况(即 key 是形如(1:2,3:4)这样的):
3. 若某 key 的 start, stop 中存在 None，则直接将值为 None 的 start 赋值为 0，将值为 None 的 stop 赋值为 self 的列数
4. 于是我们得到了 start, stop 中不存在 None 的 key
5. 使用列表元素依次读取的方式，对 self 中各个元素进行读取，将读到的元素放到新的 lst 中，得到成果

（二）调试

```
import minimatrix1 as mm
```

1. 第一种情况：

   ```
   A = mm.Matrix([[1,2],[3,4]])
   print(A[0,1])
   ```

   结果是：2

2. 第二种情况：

   ```
   A = mm.Matrix([[1,2,3],[3,4,5]])
   ```

print(A[0:1, 1:3])
结果是：[[2 3]]

3. 第三种情况：

A = mm.Matrix([[1,2,3],[3,4,5]])
print(A[:, :2])
结果是：
[[1 2]
 [3 4]]

第十个函数：矩阵修改

```
def __setitem__(self, key, value):
    if isinstance (key[0], int) and isinstance(key[1], int):
        i, j = key
        self.data[i][j] = value
    elif isinstance(key[0], slice) and isinstance(key[1], slice):
        start_row, stop_row, step_row = key[0].indices(len(self.data))
        start_col, stop_col, step_col = key[1].indices(len(self.data[0]))
        for i in range(start_row, stop_row):
            for j in range(start_col, stop_col):
                self.data[i][j] = value.data[i - start_row][j - start_col]
```

（一）功能说明

1. 第一种情况：如果要对矩阵中的某个元素进行修改。则用 isinstance（）函数判断 key 中的两个元素均为 int 类型。在这种情况下，将 key 中的两个数分别记为 i 和 j，并把矩阵中（i，j）位置的元素修改为我们想要的 value 值。这样就完成的矩阵单个元素的修改。
2. 第二种情况：如果要对矩阵中的一段序列进行修改。则用 isinstance（）函数判断 key 中的两个元素均为 slice 类型。利用 indices 方法结合数据长度，准确算出切片在行、列方向的起始、结束索引及步长。随后的嵌套 for 循环，按照这些范围，把 value.data 对应位置元素赋值给 self.data，以此完成按特定切片规则更新 self.data 中元素的操作，实现有选择的赋值。

（二）调试

```
import minimatrix1 as mm
```

1. 第一种情况：修改单个元素。

```
A =mm. Matrix(data=[
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11]
])
A[1, 2] = 0
print(A)
```

结果是：[[0 1 2 3]
 [4 5 0 7]
 [8 9 10 11]]

2. 第二种情况：修改一段序列。

```
A = mm.Matrix(data=[
    [0, 1, 2, 3],
    [4, 5, 6, 7],
    [8, 9, 10, 11]
])
A[0:2, 1:3] = mm.Matrix(data=[[0, 0], [0, 0]])
print(A)
```

结果是：[[0 0 0 3]
[4 0 0 7]
[8 9 10 11]]

第十一个函数：矩阵 n 次幂

```
def __pow__(self, n):
    if self.dim[0] != self.dim[1]:
        raise MatrixError("输入矩阵不是方阵，无法求幂！")
    if n == 0:
        return Matrix.I(self.dim[0])
    i = n - 1
    b = self.copy()
    while i > 0:
        b = Matrix.dot(self, b)
        i -= 1
    return b
```

（一）功能说明

先判断是不是方阵，不是则报错

如果是方阵，再判断 n 是否为 0

如果为 0，则返回（n,n）单位阵

如果不是 0，则先用 copy 函数，复制一个内容与 self 相同，但是 data 和其中列表地址均不同的 Matrix 对象，使用 while 循环和 dot 函数，不断相乘

（二）调试

```
import minimatrix1 as mm
ma1 =   mm.Matrix([[1,1],[1,1]],None)
print(ma1.__pow__(3))
```

结果是:

[[4 4]
[4 4]]

第十二个函数：矩阵加法

```
def __add__(self, other):
```

```
    if self.dim[0] != other.dim[0] and self.dim[1] != other.dim[1]:
        raise MatrixError("矩阵维数不匹配，无法相加减！")
    sum = [[0 for _ in range(self.dim[1])] for _ in range(self.dim[0])]
    for i in range(self.dim[0]):
        for j in range(self.dim[1]):
            x0 = self.data[i][j] + other.data[i][j]
            sum[i][j] = x0
    return Matrix(sum)
```

（一）功能说明

1. 先判断相加的矩阵的维数，若不同，则直接报错
2. 若维数相同，则同时按行读取每列的两个相加元素的值，对其加和，并加入新的 lst 中，得到结果

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
B = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(A.__add__(B))
```

结果是:

```
[[2 4 6]
 [8 10 12]]
```

第十三个函数：矩阵减法

```
def __sub__(self, other):
    other1 = Matrix(None, (self.dim[0], self.dim[1]))

    other1.data = [[-x for x in other.data[i]] for i in range(other.dim[0])]
    return self + other1
```

（一）功能说明

先对充当减数的矩阵中的每个元素取负，将得到的新矩阵与充当被减数的矩阵相加，得到结果

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 32], [4, 5, 6]])
B = mm.Matrix(data=[[1, 2, 38], [6, 5, 6]])
print(A.__sub__(B))
```

结果是:

```
[[0 0 -6]
 [-2 0 0]]
```

第十四个函数：矩阵对应置元素相乘

```
def __mul__(self, other):
    if len(self.data) != len(other.data) or len(self.data[0]) != len(other.data[0]):
        raise MatrixError("矩阵维数不匹配，无法将对应位置元素相乘！")
    else:
        result_data = [[0 for _ in range(len(self.data[0]))] for _ in range(len(self.data))]
        for i in range(len(self.data)):
            for j in range(len(self.data[0])):
                result_data[i][j] = self.data[i][j] * other.data[i][j]
        return Matrix(data=result_data)
```

（一）功能说明

1. 判断两个矩阵能否对应位置元素相乘：若两个矩阵形状相同，即行数和列数均相同，则可以进行矩阵对应位置元素乘法。否则就会返回“矩阵维数不匹配，无法将对应位置元素相乘！”。
2. 定义嵌套列表 result_data：对应位置元素相乘后的矩阵的形状与原来两个矩阵的形状相同。所以初始化一个嵌套列表 result_data，代表对应位置元素相乘后的矩阵，且矩阵中的每一个元素的初始值都是 0。
3. 乘法运算：让嵌套列表 result_data（i，j）位置处的元素的数值等于第一个列表（i，j）位置处的元素与第二个列表（i，j）位置处的元素的乘积。

（二）调试

```
import minimatrix1 as mm
a = mm.Matrix(data=[[1, 2]])
b = mm.Matrix(data=[[3, 4]])
result = a.__mul__(b)
```

结果是：[[3 8]]

第十五个函数：矩阵的元素数

```
def __len__(self):
    return self.dim[0] * self.dim[1]
```

（一）功能说明

利用 dim 直接求得

（二）调试

```
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(len(A))
```

结果是：6

第十六个函数：矩阵的字符串表示

```
def __str__(self):
```

```
    if self.data == None and self.dim == None:
        raise MatrixError("请输入参数！")
    else:
        result = "["
        first_row = True
        for row in self.data:
            if not first_row:
                result += " \n "
            else:
                first_row = False
            result += "["

            for item in range(len(row)):
                if item < len(row) - 1:
                    result += str(row[item]) + " "
                else:
                    result += str(row[item])
            result += "]"
        result += "]"
        return result
```

（一）功能说明

1. 先创建字符串 result，加入 lst[0]中每个元素的 str()形式并在各个字符串之间加上空格
2. 第一行元素加完之后，后续各行的第一格加上空格以实现元素对齐，其余同上
3. 此外，我们要在每行和每列的起止处加上左括号与右括号，并在矩阵输入完毕后加上右括号收尾，得到所求字符串

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(A)
```

结果是:

```
[[1 2 3]
 [4 5 6]]
```

第十七个函数：求行列式

做法一：不直接使用高斯函数的版本

```
def det(self):
    if len(self.data) != len(self.data[0]):
        raise MatrixError("输入矩阵不是方阵，无法求行列式！")
    else:
        if len(self.data) == 1:
            return self.data[0][0]
```

```
        else:
            result = 0
            for i in range(len(self.data)):
                sub_matrix_data = [row[:i] + row[i + 1:] for row in self.data[1:]]
                sub_matrix = Matrix(data=sub_matrix_data)
                result += ((-1) ** i) * self.data[0][i] * sub_matrix.det()
            return result
```

（一）功能说明

1. 判断矩阵是否是方阵：如果矩阵不是方阵，则它没有行列式，会返回"输入矩阵不是方阵，无法求行列式"。如果是方阵，才可以进行下一步的运算。
2. 递归终点：如果方阵的行数为 1，则它的元素的数值即为行列式的值。
3. 初始化变量：令 result 的初始值为 0。
4. 行列式计算：对于第一行的元素，让它们分别乘上其对应的代数余子式（除了其所在行和列的矩阵的行列式）。对于其代数余子式，则可以一次次递归，直到递归终点。

（二）调试

```
import minimatrix1 as mm
matrix = mm.Matrix(data = [[1, 2, 3], [3, 4, 6], [3, 6, 6]])
determinant = matrix.det()
print(determinant)
```

结果是：6

做法二：使用高斯函数

```
def det(self):
    if self.dim[0] != self.dim[1]:
        return "Error!"
    else:
        a = self.gauss().data
        pro = 1
        for i in range(len(a)):
            pro *= a[i][i]
        return pro
```

（一）功能说明

1. 判断矩阵是否是方阵：如果矩阵不是方阵，则它没有行列式，会返回"Error!!!"。如果是方阵，才可以进行下一步的运算。
2. 调用高斯函数：高斯函数会把矩阵变成一个行最简形矩阵。
3. 初始化变量：令 pro 的初始值为 1。
4. 行最简形矩阵的对角线元素的乘积即为行列式的值。

（二）调试

```
import minimatrix1 as mm
matrix = mm.Matrix(data = [[1, 2, 3], [3, 4, 6], [3, 6, 6]])
```

```
determinant = matrix.det()
print(determinant)
```
结果是：6

第十八个函数：求逆矩阵

做法一：不直接使用高斯函数的版本

```
def inverse(self):
    if self.dim[0] != self.dim[1]:
        raise MatrixError("输入矩阵不是方阵，无法求逆！")
    else:
        c = self.copy()
        a = c.data
        b = [0] * self.dim[0]
        for i in range(self.dim[0]):
            b[i] = 1
            a[i].extend(b)
            b[i] = 0
        for i in range(self.dim[0]):
            if a[i][i] == 0:
                for j in range(i, self.dim[0]):
                    if a[j][i] != 0:
                        a[i], a[j] = a[j], a[i]
                    else:
                        raise MatrixError("输入矩阵行列式为零，逆矩阵不存在！")
            elif a[i][i] != 1:
                d = a[i][i]
                for j in range(2 * self.dim[0]):
                    a[i][j] /= d

            for k in range(self.dim[0]):
                if k != i:
                    d = a[k][i]
                    for j in range(2 * self.dim[0]):
                        a[k][j] -= a[i][j] * d
    A = Matrix(a)
    return A[:, self.dim[0]:]
```

（一） 功能说明

1. 首先，进行前置判断，若矩阵的行数与列数不相等，直接返回错误提示，因为只有方阵才有逆矩阵。接着，通过 c = self.copy()复制原矩阵，获取其数据存于 a，并初始化 b 用于后续构建增广矩阵的操作。通过循环为原矩阵每行扩充对应元素来构造增广矩阵（右侧扩充单位矩阵对应元素）。
2. 然后进入核心的消元处理循环，先是检查主对角线元素，若为 0 则尝试与下方行交换

来获取非零主元，若找不到非零主元所在行则返回错误；若主元不为 1 则进行归一化处理。之后通过循环对非当前行依据主元做消元操作，逐步将增广矩阵左边化为单位矩阵。

3. 最后，利用 Matrix 类构造新矩阵 A，并返回其右侧部分（对应原增广矩阵中单位矩阵位置处），即所求的逆矩阵。
4.

（二）调试

```
import minimatrix1 as mm
ma2 = mm.Matrix([[1,4,2],[1,1,2],[1,2,1]],(3,3))
print(ma2.inverse())
```

结果是：

```
[[-1.0 0.0 2.0]
 [0.3333333333333333 -0.3333333333333333 -0.0]
 [0.33333333333333337 0.6666666666666666 -1.0]]
```

做法二：使用高斯函数

```
def inverse(self):
    if self.dim[0] != self.dim[1]:
        return "Error!!!"
    elif self.det() == 0:
        return "Error!!!"
    else:
        zr_lst = [[0 for _ in range(self.dim[1])] for _ in range(self.dim[0])]
        for i in range(len(zr_lst)):
            zr_lst[i][i] = 1
        sum = Matrix.concatenate([self,Matrix(zr_lst)],0)
        sum1 = sum.gauss()
        for i in range(sum1.dim[0]):
            for j in range(sum1.dim[1]-1,-1,-1):
                sum1.data[i][j] /= sum1.data[i][i]
        return sum1[:,self.dim[1]:]
```

（一）功能说明

1. 如果矩阵不为方阵，则不能求逆，直接返回 Error!!!

2、 如果矩阵为方阵，则先设置一个（A E）的新矩阵

3. 对此矩阵使用高斯消元法：

每次都从第 i 行第 i 列的元素开始处理，如果它为 0，则找其他行第 i 列有没有不为 0 的元素，默认从第一行第一列开始，前面的行后确定，不再移动，所以直接在下面的行中找，如果不能在这一列找到不为 0 的元素，则最终 A 的秩会小于 n，不能求逆，等价于行列式为 0，抛出异常。

再去判断该元素是否为 1，若不为 1，则通过行变换变成 1

处理好后，用消法变换使其他行的这一列变为 0

最终运用矩阵剪切输出 A 的逆

Debug 经验：改变的时候不能在循环中最好不要直接对矩阵中的多个元素进行操作，会同

时改变，导致循环中后面除的并不是原来的数，应该先将这个值赋值给一个单独的变元；注意行数和列数是谁的。

（二）调试

```
import minimatrix1 as mm
ma2 = mm.Matrix([[1,4,2],[1,1,2],[1,2,1]],(3,3))
print(ma2.inverse())
```

结果是:

```
[[-1.0 0.0 2.0]
 [0.3333333333333333 -0.3333333333333333 -0.0]
 [0.33333333333333337 0.6666666666666666 -1.0]]
```

第十九个函数：矩阵的秩

```
def rank(self):
    lst = self.gauss().data
    zero_row = 0
    flag = True
    for i in range(len(lst) - 1, -1, -1):
        for j in range(len(lst[0])):
            if not math.isclose(lst[i][j], 0):
                flag = False
                break
        if not flag:
            return len(lst) - zero_row
        else:
            zero_row += 1

    return len(lst) - zero_row
```

（一）功能说明

1. 函数首先调用了 self.gauss()，推测是利用之前定义的高斯消元相关函数对矩阵进行变换，获取处理后的矩阵数据存于 lst 中。接着初始化了 zero_row 用于统计全零行的数量，flag 用于标记当前行是否全零。
2. 随后通过从矩阵最后一行开始向前遍历的 for 循环，内层再嵌套循环遍历每行元素，利用 math.isclose 判断元素是否接近 0，若存在非零元素则将 flag 设为 False 并跳出内层循环。若 flag 为 False，意味着当前行不全零，此时返回矩阵总行数减去已统计的全零行数量，以此来确定秩；若 flag 为 True，说明当前行为全零行，全零行数量加 1。

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix(data=[[1, 2, 3], [4, 5, 6]])
print(A.rank())
```

结果是：2

第二十个函数：单位矩阵

```
def I(n):
    result = Matrix(None, (n, n))
    for i in range(n):
        result.data[i][i] = 1
    return result
```

（一）功能说明

1. 设置一个 n 维方阵 result，使其每个元素都是 0（这是最开始函数定义的，如果 data 为 None，只有 dim，则为全 0 矩阵）。
2. 把所有对角线元素赋值为 1。

（二）调试

```
import minimatrix1 as mm
n = 5
print(mm.Matrix.I(n))
```

结果是:

```
[[1 0 0 0 0]
 [0 1 0 0 0]
 [0 0 1 0 0]
 [0 0 0 1 0]
 [0 0 0 0 1]]
```

第二十一个函数：构造矩阵

```
def narray(dim, init_value=1):
    b = init_value   # 21
    lst = list(dim)
    lst.reverse()
    for i in lst:
        a = [b for _ in range(i)]
        b = a
    return Matrix(b, None, init_value)
```

（一）功能说明

narray 返回一个 matrix，维数为 dim，初始值为 init_value
narray 是 dim[0]个列表中嵌套了 dim[1]个列表，以此类推
所以先将 dim 变成 list type，然后取反，然后从里到外通过循环依次构建

（二）调试

```
import minimatrix1 as mm
a = mm.Matrix.narray([4, 5], 1)
```

print(a)
结果是:
[[1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]
 [1 1 1 1 1]]

第二十二个函数：构造矩阵

```
def arange(sa, so, se):
    lst = [[]]
    for i in range(sa, so, se):
        lst[0].append(i)
    return Matrix(lst)
```

（一） 功能说明
1. 初始化了一个包含一个空列表的列表 lst = [[]]。
2. 通过 for 循环，按照给定的范围（range(sa, so, se)）依次将符合条件的数值添加到内部的那个空列表中（lst[0].append(i)）
3. 最后把这个包含数值列表的 lst 作为构造 Matrix 对象的参数返回，完成了将生成的数值序列转换为特定数据结构对象的操作。

（二）调试
```
import minimatrix1 as mm
A = mm.Matrix([[1,2,3],[3,4,5]])
print(mm.Matrix.arange(1,3,1))
```
结果是：[[1 2]]

第二十三个函数：全 0 矩阵

```
def zeros(dim):
    return Matrix.narray(dim, 0)
```

（一）功能说明
1. 调用 narray 函数：返回一个维数为 dim 的矩阵。
2. 把 init_value 赋值为 0，使它变成一个全 0 矩阵。

（二）调试
```
import minimatrix1 as mm
dim = (3, 4)
print(mm.Matrix.zeros(dim))
```
结果是:
[[0 0 0 0]
 [0 0 0 0]

[0 0 0 0]]

第二十四个函数：全 0 矩阵（形状相同）

```
def zeros_like(matrix):
    a = matrix.data
    lst = [len(a)]
    while type(a[0]) == list:
        a = a[0]
        lst.append(len(a))
    dim = tuple(lst)
    return Matrix.narray(dim, 0)
```

（一）功能说明

1. 首先，通过 a = matrix.data 获取了输入矩阵内部所存储的数据部分，这暗示着 matrix 对象内部可能是通过 data 属性来管理实际的数值数据。
2. 接着，初始化了一个列表 lst 并将 a 的长度（也就是矩阵最外层维度的长度）放入其中（lst = [len(a)]）。
3. 然后，通过一个 while 循环不断深入判断，如果 a 的第一个元素依然是列表类型（意味着还有内层维度），就更新 a 为其第一个元素（a = a[0]），同时把这个内层维度的长度添加到 lst 中（lst.append(len(a))）。这样，经过循环后，lst 就完整地记录了矩阵各个维度的长度信息.
4. 最后将其转换为元组 dim = tuple(lst)，并调用 Matrix.narray 方法，传入维度信息 dim 和用于初始化元素为 0 的数值 0，从而创建出与输入 matrix 维度相同的全零矩阵并返回。

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix([[1,2,3],[3,4,5]])
print(mm.Matrix.zeros_like(A))
```

结果是:

```
[[0 0 0]
 [0 0 0]]
```

第二十五个函数：全 0 矩阵

```
def ones(dim):
    return Matrix.narray(dim, 1)
```

（一）功能说明

借助 Matrix 类里的 narray 方法实现。参数 dim 用于指定维度相关信息，返回的矩阵所有元素被初始化为 1。

（二）调试

```
import minimatrix1 as mm
print(mm.Matrix.ones((2,3)))
```
结果是:
```
[[1 1 1]
 [1 1 1]]
```

第二十六个函数：全 1 矩阵

```
def ones_like(matrix):
    a = matrix.data
    lst = [len(a)]
    while type(a[0]) == list:
        a = a[0]
        lst.append(len(a))
    dim = tuple(lst)
    return Matrix.narray(dim, 1)
```

（一）功能说明

1. 获取输入 matrix 的数据部分赋值给 a。
2. 将 a 的长度放入列表 lst 中。
3. 通过 while 循环，只要 a[0]的类型是列表（意味着还有内层维度），就更新 a 为其第一个元素，并把新的长度添加进 lst，如此便得到了表示维度的信息。
4. 将 lst 转换为元组 dim，并调用 Matrix.narray 方法，传入维度信息和值 1，生成并返回一个对应维度且元素全为 1 的新矩阵。

（二）调试

```
import minimatrix1 as mm
A = mm.Matrix([[1,2,3],[3,4,5]])
print(mm.Matrix.ones_like(A))
```
结果是:
```
[[1 1 1]
 [1 1 1]]
```

第二十七个函数：随机矩阵

```
def nrandom(dim):
    if len(dim) == 1:
        return random.randint(1, 100)
    elif len(dim) == 2:
        new = [[random.randint(1, 100) for _ in range(dim[1])] for _ in range(dim[0])]
    else:
        new = [[Matrix.nrandom(dim[1:]).data] for _ in range(dim[0])]
    return Matrix(new)
```

（一） 功能说明

返回一个维数为 dim 的随机 narray

1．dim 只有 1 个数的时候比较特殊，单独生成

2. dim 有 2 个数的时候，为基本情况，用 random.randint(1,100)随机生成 1-100 中的整数作为矩阵中的元素

3. dim 有多于 2 个数的时候，通过迭代生成想要的矩阵

（二）调试

import minimatrix1 as mm

1. 第一种情况：

print(mm.Matrix.nrandom((2, 3)))

结果是：

[[71 53 81]
 [57 40 58]]

2. 第二种情况：

print(mm.Matrix.nrandom((1, 2, 3)))

结果是：

[[[[65, 98, 11], [17, 8, 21]]]]

第二十八个函数：随机矩阵（形状相同）

```
def nrandom_like(matrix):
    a = matrix.data
    lst = [len(a)]
    while type(a[0]) == list:
        a = a[0]
        lst.append(len(a))
    dim = tuple(lst)
    return Matrix.nrandom(dim)
```

（一） 功能说明

1. 首先获取 matrix 的内部数据部分（a = matrix.data），接着初始化 lst 并记录最外层维度长度（lst = [len(a)]）。
2. 然后通过 while 循环，若内层元素仍为列表，就更新 a 并添加对应长度到 lst 中，以此获取完整维度信息（dim = tuple(lst)）。
3. 最后调用 Matrix.nrandom 方法，传入 dim 生成与原 matrix 同维度的随机矩阵并返回。
4.

（二）调试

import minimatrix1 as mm

A = mm.Matrix([[1,2,3],[3,4,5]])

print(mm.Matrix.nrandom_like(A))

结果是：

[[43 52 44]
 [16 25 32]]

第二十九个函数：矩阵拼接

```
def concatenate(matrices, axis=0):
    if not matrices:
        return None
    if axis == 1:
        col_num = matrices[0].dim[1]
        for matrix in matrices[1:]:
            if matrix.dim[1] != col_num:
                raise MatrixError("输入矩阵形状不对应，无法进行拼接！")
        new_data = []
        for matrix in matrices:
            new_data.extend(matrix.data)
        return Matrix(data=new_data)
    elif axis == 0:
        row_num = matrices[0].dim[0]
        for matrix in matrices[1:]:
            if matrix.dim[0] != row_num:
                raise MatrixError("输入矩阵形状不对应，无法进行拼接！")
        new_data = [[] for _ in range(row_num)]
        for matrix in matrices:
            for i in range(row_num):
                new_data[i].extend(matrix.data[i])
        return Matrix(data=new_data)
    else:
        raise MatrixError("请输入有效的维数！")
```

（一）功能说明

1. 接收矩阵列表 matrices 及可选的轴参数 axis（默认值为 0）。若 matrices 为空，则直接返回 None。

2. 当 axis 为 1 时，先获取第一个矩阵的列数，随后遍历其他矩阵，若有矩阵列数与之不符就返回 "输入矩阵形状不对应，无法进行拼接！"。若列数一致，创建空列表 new_data，将各矩阵的数据依次扩展进去，最后返回以此构建的新矩阵。

3. 若 axis 为 0，先取第一个矩阵行数，检查后续矩阵行数与之是否相同，不同则返回 "输入矩阵形状不对应，无法进行拼接！"。若行数匹配，按行数创建二维空列表 new_data，接着遍历矩阵，将每行对应元素扩展进 new_data 相应行中，最终返回构建好的新矩阵。而当 axis 为其他值时，返回 "请输入有效的维数！"，表示不符合拼接规则。

（二）调试

```
import minimatrix1 as mm
```

1. 第一种情况，横向拼接

```
A, B = mm.Matrix([[0, 1, 2]]), mm.Matrix([[3, 4, 5]])
result_0 = mm.Matrix.concatenate((A, B))
```

print(result_0)
结果是：[[0 1 2 3 4 5]]

2. 第一种情况，纵向拼接
result_1 = mm.Matrix.concatenate((A, B, A), axis=1)
print(result_1)
结果是：
[[0 1 2]
 [3 4 5]
 [0 1 2]]

第三十个函数：矩阵拼接

```
def vectorize(self, func):
    a = Matrix.copy(self)
    b = a.data
    for i in range(a.dim[0]):
        for j in range(a.dim[1]):
            b[i][j] = func(b[i][j])
    return Matrix(b)
```

（一）功能说明

1. 首先通过 Matrix.copy(self) 复制了一份输入的矩阵，确保不会修改原始矩阵的数据，将副本赋值给 a ，再获取其内部数据部分 b 。
2. 接着使用两层嵌套的 for 循环遍历矩阵的每一行（外层循环）和每一列（内层循环），在循环中针对每个元素 b[i][j] 调用传入的函数 func 进行处理，将处理后的结果重新赋值给该元素。
3. 最后把处理完的包含新元素值的 b 重新构建为 Matrix 类对象并返回，完成对整个矩阵元素按特定规则的批量处理。

（二）调试

```
import minimatrix1 as mm
a = mm.Matrix([[1, 2, 3],[2, 3, 1]], (3, 2))
def func(x):
    return x + 1
print(a.vectorize(func))
```

结果是：

```
[[2 3 4]
 [3 4 2]]
```