

# 数组化编程基础

魏华祎、王鹏祥

weihuayi@xtu.edu.cn

湘潭大学 • 数学与计算科学学院

July 15, 2020

## 目录

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

#### 4 稀疏矩阵

## 数组 (Array)

数组是一组同种类型标量数值的有序集合, 它有**维度**、**轴 (axis)** 及 **形状 (shape)** 等基本属性, 与数学中的向量、矩阵和高阶张量等概念相对应。

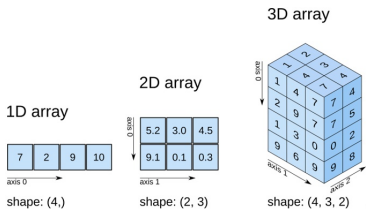


Figure: 多维数组。

### Remark

偏微分数值解中的很多算法, 如有限元、有限体积、有限差分等算法的核心数据结构就是数组, 核心操作也是对数组的操作。

- 数组化编程的基础思想是把运算操作一次性应用到整个数据集上。
- 这种编程模型允许程序员,以整个数据集合作为**思考和操作对象**,而不必求助于单个**标量操作**的**显式循环**。
- 肯尼斯·艾佛森(Kenneth E. Iverson, 1920 年 12 月 17 日 -2004 年 10 月 19 日)是一位计算机科学家,最重要的贡献是开发了 APL 语言。1979 年他因对数学表达式和编程语言理论的贡献而得到图灵奖。
- 他和许国华 (Roger Hui)(出生于香港后去加拿大)发明了 J 语言。

## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

- ☐ C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

- ☐ Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 Single Instruction, Multiple Data (SIMD).
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.

## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

- C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

- Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 Single Instruction, Multiple Data (SIMD).
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.

## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

□ C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

□ Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 Single Instruction, Multiple Data (SIMD).
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.



## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

□ C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

□ Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 Single Instruction, Multiple Data (SIMD).
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.

## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

- ☐ C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

- ☐ Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 **Single Instruction, Multiple Data (SIMD)**.
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.

## 数组化编程 (Array Programming)

- 标量化编程与数组化编程对比

- ☐ C 中的两矩阵相加

```
for (i = 0; i < n; i++)  
    for (j = 0; j < n; j++)  
        c[i][j] = a[i][j] + b[i][j];
```

- ☐ Python 中的两矩阵相加

```
c = a + b
```

- 数组化编程对应计算机科学中的 **Single Instruction, Multiple Data (SIMD)**.
- 数组化编程对应高等代数中的向量、矩阵和张量的运算, 因此高等代数给我们提供数组化编程的思维工具.

## 4 稀疏矩阵

## 1 数组化编程

## 2 Numpy

- 简介
- 多维数组对象
- 多维数组的数据类型
- 多维数组的轴及相关属性
- 多维数组的创建方法
- 多维数组元素的索引
- 多维数组的视图
- 多维数组广播
- 多维数组的组装

### 3 爱因斯坦求和

## 4 稀疏矩阵

### Python code 1 导入 Numpy 模块

```
1 import numpy as np
2 print(np.__version__)
3
4 out:
5 1.18.1
```

Python 是一种解释性的动态类型语言,可自动化进行程序内存的分配和回收。这大大减轻了用户编程的负担,并带来了极大的编程灵活性。



Figure: 鱼和熊掌不可兼得!

- Python 是一种动态类型的语言, 变量的类型可以改变。  
x = 4  
x = "four"
- C 是一种静态类型的语言, 变量类型确定后就不能再改变。

$$x = 4$$

x = "four"

```
int x = 4;
```

```
x = "four"; // 错误
```



标准的 Python 是由 C 语言实现的, 它的**整数对象**除了要表示的整数, 还存储了很多其它信息, 它的具体实现代码如下:

```

struct _longobject {
    // 引用计数器, 辅助 Python 解释器分配和回收内存.
    long ob_refcnt;
    PyTypeObject *ob_type; // 变量类型.
    size_t ob_size; // 数组 ob_digit 的长度
    long ob_digit[1]; // Python 变量实际表示的整数
};

```

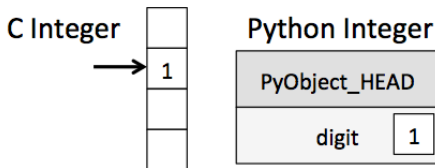


Figure: C 整数和 Python 整数的内存比较。

## Python code 2 Numpy 数组和 Python 列表

```
1 a = np.array([1, 2, 3, 4, 5, 6, 7, 8]) # Numpy 数组
2 l = [1, 2, 3, 4, 5, 6, 7, 8] # Python 列表
```

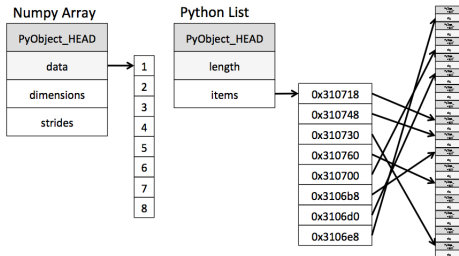


Figure: Numpy 数组和 Python 列表的内存比较。

## Python 列表对象和 Numpy 数组对象有不同之处是什么?

- **Numpy** 数组对象包含了一个指向一块连续内存的指针, 数组的所有数据就存储在这块内存中, 这样不灵活, 但数据的存储和操作更高效。
- **Python** 列表对象包含一组指针, 每个指针又指向一个 **Python** 的对象, 这些对象又各自包含自己的数据和类型信息, 这样带来极大的灵活性, 但在数据存储和操作上损失了效率。

## Remark

- 事物的尺度和规模。
- 不同尺度和规模下,理解处理问题的策略和方法都会不同。

## Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

Numpy 提供一个多维数组对象 ndarray, 它是同种类型数据的一个有序集合。下面给出一个 Numpy 数组的创建示例:

### Python code 3 用 Python 列表来创建数组

```
1 import numpy as np # 导入 numpy
2 a = np.array([[0, 1], [2, 3]]) # 创建一个二维数组
3 print("a 的对象类型为:", type(a)) # 打印变量 a 指向的对象类型
4 print(a)
5
6 out:
7 a 的对象类型为: <class 'numpy.ndarray'>
8 [[0, 1],
9  [2, 3]]
```

## 多维数组对象:np.ndarray

- 计算机内存是一个线性结构。
- 那么多维数组对象 ndarray 中的数据,在内存中是如何存储的?
- 要实现数组中元素的访问和操作,ndarray 应该有什么样的属性?

在计算机内存中,上面的二维数组 **a** 是以一行接一行的方式存储在一块连续的内存当中,这种存储方式称为行优先 (row-major) 存储。

多维数组对象

## 多维数组对象: np.ndarray

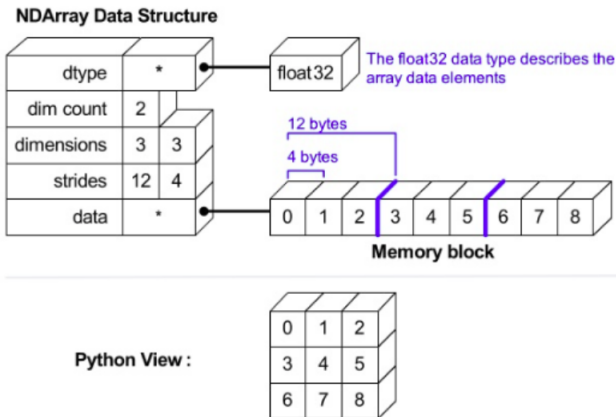


Figure: np.ndarray 的内存布局与视图。

**Remark**

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵



## 多维数组的数据类型: dtype

多维数组是用来存储同种类型数据的, 同种类型的数据占用内存的大小是固定的。多维数组中的数据类型属性 **dtype** 规定了数组中每个数据占用内存的大小, 科学计算中常用的数据类型有:

np.bool\_, 长度 8 位, 1 个字节

np.int\_, 长度 64 位, 8 个字节

np.float64, 长度 64 位, 8 个字节

np.complex128, 长度 128 位, 16 个字节

.....

多维数组的数据类型

## 多维数组的数据类型: dtype

---

### Python code 4 创建不同数据类型的数组

---

```
1 import numpy as np
2 a = np.array([0, 1], dtype=np.bool_)
3 b = np.array([12, 13], dtype=np.int_)
4 c = np.array([3.5, 4.5], dtype=np.float64)
5 d = np.array([1.0+2.0j, 3.0+4.0j], dtype=np.complex128)
6 print(a.dtype)
7 print(b.dtype)
8 print(c.dtype)
9 print(d.dtype)
```

---

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 多维数组的轴(axis)

轴 (axis) 是 Numpy 多维数组对象涉及到的一个重要概念。

- 1 维数组有 1 个轴, 2 维数组有两个轴, n 维数组有 n 个轴。
- 数组的轴是有顺序的, 编号从 0 开始。
- 数组的每一轴都是有长度的。

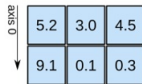
1D array



axis 0 →

shape: (4,)

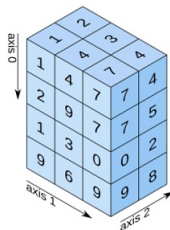
2D array



axis 1 →

shape: (2, 3)

3D array



shape: (4, 3, 2)

Figure: 多维数组的轴。

## 多维数组的 **shape** 和 **strides** 属性

---

### Python code 5 创建形状为 (3, ) 的一维数组 a

---

```
1 import numpy as np
2 a = np.array([1, 2, 3], dtype=np.int_)
3 print(a)
4
5 out:
6 [1, 2, 3]
```

---

- a 有 1 个轴
- a.shape = (3, )
- a.strides = (8, )

## 多维数组的 **shape** 和 **strides** 属性

---

**Python code 6** 下面创建一个形状为 (2, 3) 二维数组 b

---

```
1 import numpy as np
2 b = np.array([[1, 2, 3], [4, 5, 6]], dtype=np.int_)
3 print(b)
4
5 out:
6 [[1 2 3]
7  [4 5 6]]
```

---

- b 有 2 个轴
- b.shape = (2, 3)
- b.strides = (24, 8)

## 多维数组的轴

---

### Python code 7 创建一个形状为 (4, 2, 3) 三维数组 c

---

```
1 c = np.array([
2     [[ 1,  2,  3], [ 4,  5,  6]],
3     [[ 7,  8,  9], [10, 11, 12]],
4     [[13, 14, 15], [16, 17, 18]],
5     [[19, 20, 21], [22, 23, 24]]], dtype=np.int_)
6 print(c)
7
8 out:
9 array([[[ 1,  2,  3],
10         [ 4,  5,  6]],
11        [[ 7,  8,  9],
12         [10, 11, 12]],
13        [[13, 14, 15],
14         [16, 17, 18]],
15        [[19, 20, 21],
16         [22, 23, 24]]])
```

---

## 多维数组的轴

- c 有 3 个轴
- c.shape = (4, 2, 3)
- c.strides = (48, 24, 8)

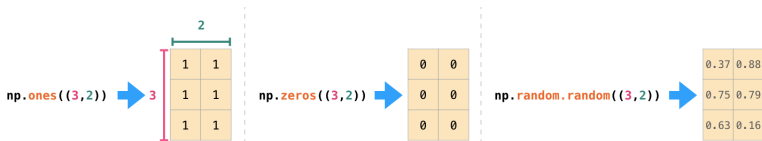


# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

多维数组的创建方法

## 多维数组对象的创建方法



Figure

- np.array
- np.arange
- np.ones
- np.ones\_like
- np.zeros
- np.zeros\_like
- .....

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 多维数组元素的索引

- 基础索引 (basic indexing), `a[1]`, `a[2, 1]`
- 切片索引 (slice indexing), `a[1:]`, `a[2:10:2]`,
- 布尔索引 (bool indexing), `a[a>0]`
- 神奇索引 (magic indexing), `a[[2, 4, 2, 5]]`
- 省略索引 (ellipsis indexing), `a[..., 1]`

## 访问修改多维数组对象中元素

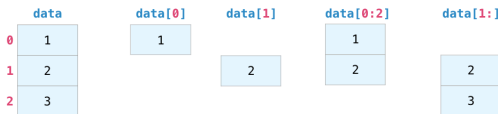


Figure: 访问一维数组中元素。

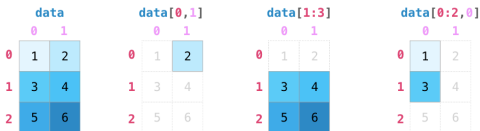


Figure: 访问二维数组中的元素。

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 视图

Numpy 为了提高内存利用率, 尽量避免大数据的拷贝操作, 在切片索引等一些操作中, 返回的并不是原始数组的一个副本 (**copy**), 而是一个视图 (**view**)。

- 多维数组的视图可以当做普通的多维数组使用。
- 但一定要注意, 修改视图数组的元素, 同时也会修改原始数组的对应元素。
- 而修改副本数组的元素, 不会影响到原始数组的对应元素。
- 在 Numpy 中, 能返回视图, 尽量返回视图, 如切片索引和省略索引返回的都是数组的视图。
- 在基于 Numpy 的科学计算程序设计中, 要时刻明确, 当前使用的数组是: 原始数组、副本数组和视图数组的哪一种。

# Outline

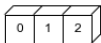
- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵



## 数组中的广播 (Broadcasting)

Numpy 中形状相同的数组可以逐元素地进行二元运算, 而**广播 (Broadcast)** 可以允许不同形状的数组 在一定的规则下进行运算。

`np.arange(3)+5`



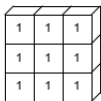
+



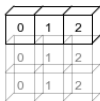
=



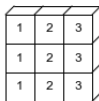
`np.ones((3,3))+np.arange(3)`



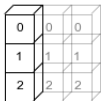
+



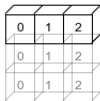
=



`np.arange(3).reshape((3,1))+np.arange(3)`



+



=

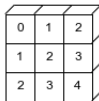


Figure: 多维数组广播运算实例。

## 广播运算规则

Numpy 的广播遵循一组严格的规则：

- 如果两个数组的维度数不相同, 那么维度数小的数组的形状将会在最左边补 1。
- 如果两个数组的形状在任何一个维度上都不匹配, 那么数组的形状会沿着维度为 1 的维度扩展以匹配另外一个数组的形状。
- 如果两个数组的形状在任何一个维度上都不匹配并且没有任何一个维度等于 1, 那么就会引发异常。

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 多维数组的组装

在科学计算中,有时我们需要创建一人数组,但我们仅知道一个数组的形状,及一些位置应该放一些什么值,这时就需要用到 Numpy 提供的**数组组装功能**。

---

### Python code 8 多维数组组装示例

---

```
1 import numpy as np
2 indices = np.array([0, 0, 1, 3, 5], dtype=np.int_)
3 data = np.array([4.5, 3.5, 6.4, 3.2, 1.0], dtype=np.float64)
4 a = np.zeros(6, dtype=np.float64)
5 np.add.at(a, indices, data)
6 # a[indices] += data
7 # np.negative.at, np.multiply.at, np.divide.at
8 # np.logical_and.at, np.logical_or.at, .....
```

---

## Numpy 程序设计的一些基本原则

- 尽量避免复制数据。
- 默认为行优先 (row-major) 存储。
- 利用视图和广播 (broadcast) 功能, 尽量避免开辟新的内存。
- 利用数组化编程 (array programming) 方式提高编写代码和程序执行的效率。

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 爱因斯坦求和:np.einsum

爱因斯坦求和 (Einstein Summation) 是一种张量运算的维度标记约定, 可以使张量运算的数学表达式更加简洁明快。

```
14 # 3: Format string. Incomplete example with 3 input args.  
15 dst = np.einsum("□,□,□->□", arg0, arg1, arg2)
```




Figure: Einstein 求和。

### Remark

np.einsum 是进行单元矩阵组装的利器! 一定要熟练掌握!

### Python code 9 矩阵转置 $B_{ji} = A_{ij}$

```
1 import numpy as np
2 a = np.arange(0,9).reshape(3,3)
3 print("a:\n", a)
4 b = np.einsum('ij->ji', a)
5 print("b:\n", b)
6
7 out:
8 a:
9 [[0 1 2]
10  [3 4 5]
11  [6 7 8]]
12
13 b:
14 [[0 3 6]
15  [1 4 7]
16  [2 5 8]]
```



## 爱因斯坦求和

---

### Python code 10 全部元素求和 $\sum_i \sum_j A_{ij}$

---

```
1 import numpy as np
2 a = np.arange(0,9).reshape(3,3)
3 print("a:\n", a)
4 b = np.einsum('ij->', a)
5 print("b:", b)
6
7 out:
8 a:
9 [[0 1 2]
10  [3 4 5]
11  [6 7 8]]
12 b: 36
```

---

## 爱因斯坦求和

某一维度求和  $\sum_i A_{ij}$

---

```
1 import numpy as np
2 a = np.arange(0,9).reshape(3,3)
3 print("a:\n", a)
4 b = np.einsum('ij->i', a)
5 print("b:", b)
6
7 out:
8 a:
9 [[0 1 2]
10  [3 4 5]
11  [6 7 8]]
12
13 b: [ 3 12 21]
```

---

# Outline

- 1 数组化编程
- 2 Numpy
  - 简介
  - 多维数组对象
  - 多维数组的数据类型
  - 多维数组的轴及相关属性
  - 多维数组的创建方法
  - 多维数组元素的索引
  - 多维数组的视图
  - 多维数组广播
  - 多维数组的组装
- 3 爱因斯坦求和
- 4 稀疏矩阵

## 简介

给定一个矩阵  $A$ , 如果其非零元个数比较少, 我们就称其为**稀疏矩阵**。

稀疏矩阵是数值计算中经常用到的矩阵类型, 如有限元、有限差分、有限体积等常用的偏微分方程数值求解方法, 最后得到的都是稀疏矩阵。因此, 学习数值计算很有必要了解这种矩阵在计算机中的存储格式。

如果矩阵的总数据量较大, 完成存储这个矩阵会有大量的 0 存储, 会浪费很多空间。因此, 我们需要为稀疏矩阵设计不同的存储格式。

## 简介

- 坐标格式稀疏矩阵:Coordinate format Matrix(COO)
- 压缩稀疏行稀疏矩阵:Compressed Sparse Row Matrix(CSR)
- 压缩稀疏列稀疏矩阵:Compressed Sparse Column Matrix(CSC)

---

### Python code 11 scipy.sparse 中的稀疏矩阵类

---

```
1 from scipy.sparse import coo_matrix
2 from scipy.sparse import csr_matrix
3 from scipy.sparse import csc_matrix
```

---

# COO

$$A = \begin{pmatrix} 0 & 0 & 0 & 10 \\ 21 & 0 & 33 & 0 \\ 0 & 0 & 3 & 0 \\ 12 & 1 & 0 & 4 \end{pmatrix}$$

存储格式如下：

$$data = [10, 21, 33, 3, 12, 1, 4],$$

$$I = [0, 1, 1, 2, 3, 3, 3],$$

$$J = [3, 0, 2, 2, 0, 1, 3],$$

$$A = \begin{pmatrix} 0 & 0 & 0 & 10 \\ 21 & 0 & 33 & 0 \\ 0 & 0 & 3 & 0 \\ 12 & 1 & 0 & 4 \end{pmatrix}$$
$$\begin{aligned} data &= [10, 21, 33, 3, 12, 1, 4], \\ indices &= [3, 0, 2, 2, 0, 1, 3], \\ indptr &= [0, 1, 3, 4, 7]. \end{aligned}$$

$$A = \begin{pmatrix} 0 & 0 & 0 & 10 \\ 21 & 0 & 33 & 0 \\ 0 & 0 & 3 & 0 \\ 12 & 1 & 0 & 4 \end{pmatrix}$$

SC 格式如下:

$$\begin{aligned} data &= [21, 12, 1, 33, 3, 10, 4], \\ indices &= [1, 3, 3, 1, 2, 0, 3], \\ indptr &= [0, 2, 3, 5, 7]. \end{aligned}$$