第七章 Numpy计算科学模块（六）：掩码索引

7.1 示例：统计下雨天数

> 知识点：

- 之前已经学了**单索引**和**切片索引**方式来访问数组。
- NumPy还为数组提供了更佳个性化的访问功能，通过设定条件来筛选数组元素。例如统计数组中有多少值大于某一个给定值，或者删除所有超出某些门限值的异常点。
- 本节的NumPy布尔掩码将实现上述功能。

任务：降雨统计

下面将通过一个实际案例，讲解NumPy的布尔掩码的功能。这里选用2014年西雅图市的日降水统计数据，并使用Pandas读取数据。

```
# 导包操作
import numpy as np
import pandas as pd

# 调用.read_csv()读取csv文件
data = pd.read_csv('Seattle2014.csv')
# 显示表格数据各字段(特征)信息
print(data.info())
```

- 调用 `.read_csv()` 读取csv文件，该方法的输入为 `.csv` 为扩展名的文件， `data` 为pandas对象。
- 通过 `.info()` 显示 `data` 数据字段(特征)组成：每个样本由17个特征组成。其中 `PRCP` 为降雨量，后面需要用这个特征进行降雨量统计。

```
# 使用pandas提取降雨量'PRCP'字段，赋值给NumPy数组
rainfall = data['PRCP'].values
# mm到inches的单位转换
inches = rainfall / 254
print(inches.shape)
```

`inches` 数组包含365个值，给出了从 2014 年 1 月 1 日至 2014 年 12 月 31 日每天的降水量。这里降水量的单位是英寸(inches)。为更好分析数据，首先使用Matplotlib对该数据进行可视化，生成雨天直方图。

任务：掌握直方图的使用

知识点：频率分布直方图，简称**直方图**。

- **直方图：将数据按区间进行分组，然后统计落在各分组的频数，最后进行可视化输出。**
- 直方图可以将数据的分布信息直观、形象地表示出来，清楚地展示各分组的频数分布情况，让我们能够更好了解数据的分布情况。

- 如何对数据进行分组，直接影响着直方图的最终结果。分组过少，数据就非常集中；分组过多，数据就非常分散，这就掩盖了分布的特征。

知识点： `matplotlib` 模块提供了直方图绘制方法 `.hist(x, bins=None)` 。该方法有很多个参数，这里介绍常用两个参数。

- `x` 为需要统计的数据数组。
- `bins` 为划分分组数。

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set()

# 调用matplotlib模块的直方图方法.hist
plt.hist(inches, 40);
```

任务：向量化逻辑运算的需求分析

由直方图可知：尽管人们对西雅图的印象下雨较多，但是2014年它大多数时间(超过200天)的降水量都是接近0的。

知识点：**直方图并不能很好的反映某些微观信息**。例如，一年中有多少天在下雨，这些下雨天的平均降水量是多少，有多少天的降水量超过了半英寸？不能定量分析！**如何实现上述的需要?**

- 传统方法：通过遍历，对元素进行逐一的判断。
  - **遍历所有数据，当碰到数据落在希望的区间时计数器便加 1**。
  - 虽然这种方法比较简单粗暴，但它的效率非常的低。
- NumPy方法：向量化操作。
  - 使用NumPy的**通用函数来替代循环**，以快速实现数组逐元素(element-wise)运算。
  - NumPy提供了逻辑通用函数，可高效的实现数组逐元素**逻辑比较**。
  - 可通过逻辑运算来**筛选满足特定条件的数组元素**，然后对筛选的元素进行后续的数据分析。

7.2 逻辑运算与布尔数组

7.2.1 NumPy逻辑运算的通用函数

任务：比较运算操作符

下面将介绍NumPy针对数组逻辑运算提供的通用函数。通过这些通用函数，可以实现高效的元素筛选需求。

> 知识点：比较运算操作符，实现了数组的逐元素逻辑比较操作。
>
> - NumPy提供了6种比较运算操作符： `<` 、 `<=` 、 `>` 、 `>=` 、 `!=` 、 `==` 。
> - 逻辑运算结果：**布尔数据类型的数组**(`True` 或 `False`)，它的 `.shape` 是数组广播后的结果。
> - 思考 `=` 和 `==` 的区别?

> 知识点：**布尔数组可以作为数组的掩码数组**，用于数组元素的筛选。

```
x = np.array([1, 2, 3, 4, 5])

print(x < 3)  # 小于
print(x > 3)  # 大于
print(x <= 3)  # 小于等于
print(x >= 3)  # 大于等于
print(x != 3)  # 不等于
print(x == 3)  # 等于
```

还可以利用复合表达式进行逻辑判断。

```
# 看赋值=的右边表达式
# 获得(2 * x)和(x ** 2)是否相等的逻辑比较结果
res = (2 * x) == (x ** 2)
print(res)
```

任务：逻辑通用函数

> 知识点：
>
> - 和算术运算符一样，**比较运算操作符**在 NumPy 中也是借助通用函数来实现的。
> - 例如 `x < 3` 时，NumPy内部会使用 `np.less(x, 3)` 。
> - 下表给出了比较运算符和其对应的通用函数。

运算符	对应的通用函数
==	`np.equal`
`!=`	`np.not_equal`
`<`	`np.less`
`<=`	`np.less_equal`
`>`	`np.greater`
`>=`	`np.greater_equal`

> 知识点：当遇到参与运算数组的 `.shape` 不匹配时，逻辑通用函数同样也可以使用广播机制。

```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
print(x)
#逻辑比较运算结果都是布尔数组
print(x < 6)
```

7.2.2 布尔数组(掩码)

当得到了逻辑结果的布尔数组(称之为**掩码**)，可以实现很多后续数据分析操作。

任务：统计记录的个数

> 知识点： `False` 会被解释成 `0` ， `True` 会被解释成 `1` 。

> 知识点：为了统计布尔数组中 `True` 记录的个数，可使用 `np.count_nonzero()` 函数。
>
> - 语法： `numpy.count_nonzero(a, axis=None)` 。
> - 描述：统计数组 `a` 非零元素个数。
> - 参数： `a` 为待统计数组。

- 参数： `axis` 可选。类似通用函数 `.sum()` 。缺省默认值，整个数组的非零元素个数统计；指定 `axis` 参数，沿 `axis` 聚合来非零元素个数统计。
- 返回：标量(`axis` 缺省默认值)或数组(指定 `axis` 参数)。

```python
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))

# 逻辑比较
res = (x < 6)
print(res)

# x<6元素的总数
print(np.count_nonzero(res))

# 沿第0轴，x<6的个数
print('axis=0', np.count_nonzero(res, axis=0))

# 沿第1轴，x<6的个数
print('axis=1', np.count_nonzero(res, axis=1))
```

另一种方法，可以调用 `np.sum()` 。

```python
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))

# 逻辑比较
res = (x < 6)
print(res)

# x<6元素的总数
print(np.sum(res))

# 沿第0轴，x<6的个数
print('axis=0', np.sum(res, axis=0))

# 沿第1轴，x<6的个数
print('axis=1', np.sum(res, axis=1))
```

知识点：

- `np.any(a, axis=None)` ： 检查布尔数组是否任意包含 `True` ，只要有一个 `True` 元素就返回 `True` 。

- `np.all(a, axis=None)`：检查布尔数组是否全部包含 `True`，只有所有都是 `True` 元素才返回 `True`。

```
print(x)
print('np.any(x > 8):', np.any(x > 8))  # 有没有值大于8?
print('np.any(x < 0):', np.any(x < 0))  # 有没有值小于0?
print('np.all(x < 10):', np.all(x < 10)) #是否所有值都小于10?
print('np.all(x == 6):', np.all(x == 6))  #是否所有值都等于6?
```

`np.all()` 和 `np.any()` 通过指定参数 `axis` 对布尔数组进行沿着特定坐标轴的聚合。

```
print(x)
# 沿axis=1(行)，是否所有值都小于6?
print(np.all(x < 6, axis=1))
```

任务：布尔运算符

通过比较运算符，可以实现统计所有降水量 `<4` 英寸或者 `>2` 英寸的天数，那么如果**想统计降水量同时 `<4` 英寸且 `>2` 英寸的天数该如何操作呢**？本节将介绍布尔运算符来实现上述目标。

知识点：布尔运算符

- 类似于Python，NumPy使用了**布尔通用函数**重载了4个布尔运算符：`&`、`|`、`^` 和 `~`。具体描述如下表。
- 布尔运算符的返回为**布尔数组**(元素为 `True` 和 `False`)。

运算符	对应通用函数	功能
`&`	`np.bitwise_and`	与
`|`	`np.bitwise_or`	或
`^`	`np.bitwise_xor`	异或
`~`	`np.bitwise_not`	非

7.2.3 使用布尔运算分析数据

任务：使用比较和布尔运算分析数据

下面的代码是基于“任务：降雨统计”，如果不能运行，先运行该任务代码。

```
# 复合表达式：比较运算+布尔运算的组合来构造结果布尔数组
res = (inches > 0.5) & (inches < 1)
print(res.shape, res.ndim)
#print(res)

# 统计满足条件的元素个数
res_count = np.sum(res)
print(res_count)
```

结果可知，降水量在0.5英寸到1英寸间的天数是29天。

> 知识点：**通过括号，可以控制运算符之间的优先级。**

如果修改括号的位置，那么表达式的运行结果将会出错误：

```
# 错误的逻辑
res = inches > (0.5 & inches) < 1
```

> 知识点：可以利用 **A AND B** 和 **NOT (NOT A OR NOT B)** 的等价原理(数电课应该学过)，实现逻辑等价变换。在实际编程中，建议选择简单的不易出错的表达式。

```
# 复合表达式：比较运算+布尔运算的组合来构造结果布尔数组
res = ~( (inches <= 0.5) | (inches >= 1) )
print(res.shape, res.ndim)
#print(res)

# 统计满足条件的元素个数
res_count = np.sum(res)
print(res_count)
```

```
print(inches > (0.5 & inches) < 1) # 错误的逻辑
```

利用比较运算符和布尔运算符来获得掩码数组，来分析之前那些关于天气的问题。

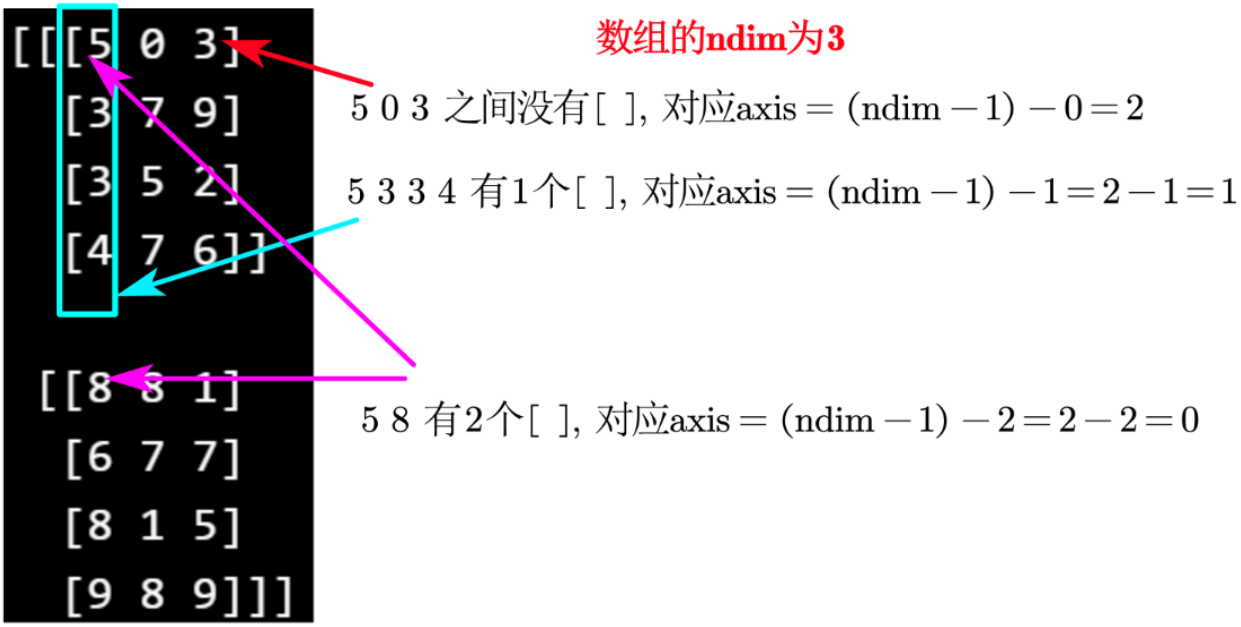
```
print("Number days without rain:", np.sum(inches == 0))
print("Number days with rain:", np.sum(inches != 0))
print("Days more than 0.5 inches:", np.sum(inches > 0.5))

print("Rainy days with < 0.1 inches:", np.sum((inches > 0) & (inches < 0.2)))
```

7.3 如何将数组元素和axis进行关联

任务：(重要)如何将数组元素和axis进行关联

> 知识点：通用函数的聚合操作依赖于 `axis` 参数。即沿指定 `axis` 对数组进行聚合操作。对于高维的数组，很难将数组元素和 `axis` 进行关联。那么如何判断 `axis` 所对应的元素？
>
> - 这里给出一个简单的方法来找出 `axis` 对应的元素：通过 `[]` 所在的层数来判断。
> - 换算公式：如果元素之间间隔n个 `[]` ，那么对应的 `axis = (.ndim-1) - n` ， `.ndim` 为数组总维度。 `axis` 的值从大到小，对应的 `[]` 是从内到外的。结合下图理解。
> - 聚合操作，就是将 `axis` 对应位置的元素进行运算。



下面例子，请仔细观察每个 `axis` 聚合对应的元素，对数据矩阵 `X` 挑几个元素，进行理论上计算，然后再看结果是否正确。另外，理论上计算每个 `axis` 聚合后得到的结果数组 `.shape` 。

```
import numpy as np
rng = np.random.RandomState(0)
X = rng.randint(10, size=(2, 4, 3))
print(X)
print('X.shape', X.shape)
print('-------np.sum(X, axis=0) 间隔2个[]的元素聚合-------')
print(np.sum(X, axis=0))
print('-------np.sum(X, axis=1) 间隔1个[]的元素聚合-------')
print(np.sum(X, axis=1))
print('-------np.sum(X, axis=2) 没有[]的元素聚合-------')
print(np.sum(X, axis=2))
```

7.4 NumPy之掩码索引

任务：将布尔数组作为掩码来筛选元素

> 知识点：逻辑判断与 `if` 的条件判断相似，它对数组中的每个元素分别做逻辑判断，满足条件的元素对应的布尔数组值为 `True` ，反之则为为 `False` 。

> 知识点：之前已在掌握了，如何**使用比较和布尔运算的组合来设置逻辑条件**。
>
> - 当获得布尔数组后，可以将得到的布尔数组作为**掩码**来选中满足条件的元素(布尔数组为 `True` 对应的元素)，并生成新的子数据集。
> - **掩码**规则：使用**掩码**来筛选元素，返回的是一个一维数组，它包含了**所有满足条件的元素值**。换句话说，所有的元素是布尔数组对应位置为 `True` 的元素。
> - **这里有个坑**：返回的元素数组为一维数组，它的 `.shape` 和原数组无关。

下面通过一个例子来讲解布尔数组是如何作为**掩码**来使用的。

```
x = rng.randint(10, size=(3, 4))
print(x)

# 布尔数组的获得
res = (x < 5)
print(res)

# 将布尔数组作为隐码(索引)来筛选元素
```

```python
# 被筛选的子数据为一维向量
new_x = x[res]
print(new_x)

# 直接使用逻辑条件作为隐码，来筛选元素
new_x1 = x[x < 5]
print(new_x1)
```

接下来，设置一些条件，来实现西雅图降水数据进行一些相关统计分析：

```python
# 为所有下雨天创建一个掩码
rainy = (inches > 0)

# 构建一个包含整个夏季日期的掩码（6月21日是第172天）
summer = (np.arange(365) - 172 < 90) & (np.arange(365) - 172 > 0)

print("Median on rainy days in 2014 (inches): ", np.median(inches[rainy]))

print("Median on summer days in 2014 (inches):  ",
np.median(inches[summer]))

print("Maximum on summer days in 2014 (inches): ", np.max(inches[summer]))

print("Median on non-summer rainy days (inches):", np.median(inches[rainy &
~summer]))
```

任务：数组索引的掩码

接下来，看一个更具有挑战性的**掩码**使用，此题目为企业笔试题。先仔细思考，后看结果分析。

```python
import numpy as np
x = np.arange(16).reshape(4,4)
print(x)

# 掩码索引
res = x[(True,False,True,False),(False,True,False,True)]
res1 = x[(True,False,True,False),:]
res2 = x[:,(False,True,False,True)]
print(res)
print(res1)
print(res2)
```

知识点：精华分析(多看几遍)

- 以 `res = x[(True,False,True,False),(False,True,False, True)]` 为例，其他的代码以此类推。
- 此题考的是数组元素的索引的**掩码**。看清楚，是对索引的**掩码**，而对非元素的**掩码**。
- 对于第1维，`(True,False,True,False)` 是布尔数组。根据**掩码**规则可知，第0和2索引位置为 `True` ，第1和3索引位置为 `False` 。返回结果为索引数组：`(0,2)` 。
- 对于第2维，`(False,True,False,True)` ，返回结果为索引数组：`(1,3)` 。
- 最后将得到的**索引**回归到**数组**，`x[(0,2),(1,3)]` 。返回位置为 `(0,1)` 和 `(2,3)` 的元素。
- 两维上的 `True` 的个数必须相同，否则就会出错。请自行验证。思考原因?

7.5 比较逻辑运算符(and,or)和(&,|)

任务：比较逻辑运算符(and,or)和(&,|)

知识点：

- `and` 和 `or` 是关键字，`&` 和 `|` 是布尔运算符。
- 语法：`A and B` ，`A or B` ，`A & B` ，`A | B` 。

相同之处：当A，B均为**布尔元素**(`True` 和 `False`)时，关键字 `and` 和 `or` 与布尔运算符 `&` 和 `|` 功能相同。

```
res = (1 > 2) and (1 < 2)
res1 = (1 > 2) & (1 < 2)
print(res, res1)
```

不同之处: 当A, B为非布尔类型的元素时。

- 适用于任何对象：`and` 和 `or` 判断**整个对象A，B是真或假**, 等价于让Python将对象当作整体进行逻辑比较。**在python中，任何非零或非空的对象都会被当作是** `True` 。
- 只适用于布尔值或整数：`&` 和 `|` 是判断每个对象中的**比特位**(只有整数才能用比特位表示)。因此，**当A，B为非整数的对象或或数组时，使用 `&` 和 `|` 会出现错误**。

- & 和 | 支持向量化运算， and 和 or 无法使用向量化运算。

```
# 虽然42和1不一样，但是逻辑结果是true
print('42 and 1:', bool(42 and 1))
print('42 and 0:', bool(42 and 0))
print("'42' and '0':", bool('42' and '0'))

print('42 or 1:', bool(42 or 1))
print('42 or 0:', bool(42 or 0))
```

```
# bin()函数将整数转换为二级制数
# 输出结果'0b'代表二级制
print(bin(42))
print(bin(59))
# 按位进行逻辑比较
print('42 | 59:', bin(42 | 59))
print('42 & 59:', bin(42 & 59))
```

```
# 非整数或布尔类型，适用布尔运算符会出错
print("'42' and '0':", bool('42' | '0'))
```

```
# NumPy通过dtype = bool声明布尔数组
A = np.array([1, 0, 1, 0], dtype = bool)
B = np.array([1, 1, 1, 0], dtype = bool)
print(A | B)
```

知识点(总结)： and 和 or 是将整个对象作为执行单位来执行布尔运算，而 & 和 | 是对一个对象中具体内容按位(二进制形式)执行布尔运算。

7.6 NumPy数据的输入输出

第8章 NumPy计算科学模块(七)：花式索引

8.1 初识花式索引

任务：NumPy数组的四大索引

> 知识点：
>
> - 单索引：根据**单个索引值**，返回**单个元素**。例如， `x[0,2]` 。
> - 切片索引：使用 `[start:stop:step]` 切片规则来索引元素，返回**一组元素**。例如， `x[:2,1:]` 。
> - 掩码索引：使用逻辑运算得到布尔数组作为**掩码**，返回掩码为 `True` 的**一维数组元素**。例如， `x[x > 0]` 。
> - 花式索引：fancy indexing，向数据数组传递**索引数组**以便批量获得**多个数组元素**。(本节学习重点)

> 多维数组的索引：**单索引、切片索引、花式索引的多维数组索引方式。**
>
> - 按**一维数组索引**的方式，**对每个维度分别使用索引**。
> - 例如， `x[0,1]` ，每个维度分别使用索引，第1维索引为 `0` ；第2维索引为 `1` ；然后NumPy使用各维度上的索引来**定位元素**，获得 `(0,1)` 坐标位置上的元素；不同维度的索引，则通过 `,` 相隔。

> 多维数组的索引：**掩码索引的多维数组索引方式。**
>
> - 一维索引和多维索引一样，只需要输入整个掩码数组，并不需要为每个维度分别输入隐码数组。 `x[x>1]` 。

```
import numpy as np
rand = np.random.RandomState(42)
# 创建二维数组
x = rand.randint(10, size=(5,5))
print(x)

print('--单索引--')
print(x[0,1])
```

```
9
10  # 第1维0~1;第2维1~end
11  print('--切片索引--')
12  print(x[:2,1:])
13
14
15  #不需要为每个维度单独索引
16  print('--掩码索引--')
17  # 构造掩码
18  mask = x<3
19  print(mask)
20  print(x[mask])
```

```
1   # 不需要为每个维度单独索引
2   print('--一维掩码索引--')
3   x = rand.randint(10, size=(5,))
4   print(x)
5
6   # 构造掩码
7   mask = x<3
8   print(mask)
9   print(x[mask])
10
11  print('--多维掩码索引--')
12  x = rand.randint(10, size=(5,3))
13  print(x)
14
15  # 构造掩码
16  mask = x<3
17  print(mask)
18  print(x[mask])
```

任务：初识花式索引

> 知识点：花式索引直观认识
>
> - **切片的瓶颈**：虽然切片可以批量获得一组数组元素，但是切片规则获得的元素必须要有一定的规律 `[start:stop:step]` 。对于无规则的批量索引，切片则无法实现。例如，一维数组，按顺序获得 `1` ，`2` ，`4` ，`0` 位置的元素。
> - 为解决上述问题，NumPy引入了花式索引机制。
> - 将**单索引**推广至**索引数组**，从而实现批量索引数组的元素。
> - 相对于单索引，花式索引使用一个**整型数组**作为**索引数组**，来获得多个数组元素。

下面通过一个例子，如何将将**单索引**推广至**花式索引**，来方便地实现数组元素的批量获取。

```
x = rand.randint(100, size=10)
print(x)
```

> **方法一：单索引**。依次使用单索引来获得元素。

```
res = [x[3], x[7], x[2]]
print(res)
```

> **方法二：花式索引**。构造索引数组，然后再将该数组作为**整体**传递给数组。

```
ind = [3, 7, 4]
res = x[ind]
print(res)
```

任务：掌握索引相关的术语

> 知识点：(重点) **索引相关的术语**。
>
> - 数据数组：用于被索引的数组。
> - 结果数组：索引之后得到的数组。
> - 索引数组：该数组的元素是索引值，必须是整形，用于索引数据数组。注意：索引值不能超过数据数组最大的索引，否则将会出错。
> - 数组的总维度：数组的 `.ndim` 属性。
> - 数组的某个维度：多维数组，需要指明索引是针对第几维或哪个 `axis` 。
> - 数组的形状：数组的 `.shape` 属性。

下面结合一个一维数组的花式索引例子，来了解术语。

```
print('--x称为数据数组--')
x = rand.randint(10, size=(5,))
print(x)

print('--数组维度和形状--')
print(x.ndim, x.shape)

```

```
print('--ind称为索引数组--')
ind = [1,2,4,0]
print(ind)

print('--x_new称为结果数组--')
print('--x[ind]为x指定第1维的索引为ind--')
x_new = x[ind]
print(x_new)
```

```
x = rand.randint(100, size=10)
print(x)

ind = np.array([[3, 7], [4, 5]])
res = x[ind]
print(res)
# 结果数组shape与索引数组一致
print('数据数组shape: ', x.shape, x.ndim)
print('结果数组shape: ', res.shape, res.ndim)
print('索引数组shape: ', ind.shape, ind.ndim)
```

8.2 多维数组的花式索引

任务：多维数据数组的单索引和切片

要理解多维数组的花式索引，先回顾多维数组的单索引和切片。

> 知识点：(回顾) 多维数组的单索引和切片。以三维数组 `x` 为例。
>
> - 多维数组的元素访问：在 `[]` 中，以 `,` 相隔不同维度的索引值，来获得数组元素。
> - **单索引**：获得一个元素。 `x[i,j,k]` ，其中 `i` ， `j` 和 `k` 为索引值分别对应 `axis=0` 、 `axis=1` 和 `axis=2` 的坐标值。例如， `x[2,1,3]` 。
> - **切片**：获得一组元素。而单索引本质上是切片的一个特例。例如， `x[2:,:,:3]` 。
> - **单索引和切片**的组合：有些 `axis` 使用单索引，有些 `axis` 使用切片。例如， `x[2,1:,:3]` 。

```
# 创建一个三维数组
x = np.arange(36).reshape((3, 3, 4))
print('数据数组: ', x.ndim, x.shape)
print(x)
```

```
5
6  # 单索引
7  print('---单索引---')
8  print(x[2,1,3])
9
10 # 切片
11 print('---切片---')
12 print(x[2:,1:,:3])
13
14 # 单索引和切片的组合
15 print('---单索引和切片的组合---')
16 # 中间的：不能被缺省
17 print(x[2,:,:3])
```

任务：多维数组的索引缺省值

> 知识点(规则)：在NumPy中，**多维数组的索引是否可以被缺省?**
>
> - 规则：**允许**数组的**高维** `axis` **索引**被缺省(被缺省的索引之后，**没有显式索引值**)。
>   - 被缺省的 `axis` 索引值默认使用切片 `:` ，即返回该 `axis` 的所有元素。
>   - 例如，三维数组 `x[2,2]` 等价于 `x[2,2,:]` 。四维数组 `x[2,2]` 等价于 `x[2,2,:,:]` 。
> - 规则：**不允许**数组的**低维** `axis` **索引**被缺省(被缺省的索引之后，**还有显式索引值**)。
>   - 例如，第1维和第3维有显式的索引，第2维的索引就不能被缺省，即两个 `,` 之间不能为空。 `x[2,:,2]` 不能缺省为 `x[2,,2]` 。

```
1  # 缺省索引
2  print('---缺省索引---')
3  # 高维索引：第3维的索引可以被缺省
4  print(x[2,2])
5
6  # 低维索引不能被缺省
7  # 出错的缺省索引，两个,之间不能为空
8  print('---出错的缺省索引---')
9  #print(x[2,,2])
```

任务：掌握结果数组总维度的演算规则

> 知识点(规则)：结果数组的总维度 .ndim 的换算公式。
>
> - 结果数组 .ndim = 数据数组 .ndim - 使用单索引的 axis 个数。
> - **切片对结果数组的维度没影响**，哪怕切片返回的元素只有一个或者为空。

```
# 创建一个三维数组
x = np.arange(36).reshape((3, 3, 4))
print('数据数组: ', x.ndim, x.shape)

print('---结果数组.ndim = 数据数组.ndim - 使用单索引的axis个数---')
x_new = x[2,1,3]
# x_new的.ndim = 数据数组.ndim - 3个axis使用单索引 = 3 - 3 =0
print(x_new)
print('x[2,1,3]: ', x_new.ndim, x_new.shape)
```

```
x_new = x[2,:,3]
# x_new的.ndim = 数据数组.ndim - 2个axis使用单索引 = 3 - 2 =1
print(x_new)
print('x[2,:,3]: ', x_new.ndim, x_new.shape)
```

```
x_new = x[2,:,:]
# 自行分析
print('x[2,:,3]: ', x_new.ndim, x_new.shape)
```

> 知识点：如果某个维度**切片返回的元素是空**，对结果数组 .ndim 并未有影响。

```python
print('--切片对结果数组.ndim没影响--')
# 第3维切片为1个元素
x_new = x[2:,:,3:]
print(x_new)
# 自行分析
print('x[2,:,3]: ', x_new.ndim, x_new.shape)

#坑：第3维切片为0个元素，结果数组为空数组[]
# 根据NumPy规则，这数组还是有维度。
x_new = x[2:,:,4:]
print(x_new)
# 自行分析
print('x[2,:,3]: ', x_new.ndim, x_new.shape)
```

任务：多维数组的花式索引

之前回顾了多维数组的单索引和切片，接下来讲下多维数组的花式索引的使用。

> 知识点：
>
> - 本质上，花式索引就是将**多个单索引组成为一个索引数组(序列)**。通过该索引数组，批量获得元素。
> - 多维数组的单索引：必须为每个 `axis` 指定索引值。例如： `x[1,2,1]` 。
> - 多维数组的花式索引：类似，也必须要对每个 `axis` 指定索引数组。例如： `x[ind1,ind2,ind3]` 。
> - 规则：如果数据数组总维度为 `.ndim` ，那就需要 `.ndim` 个之对应的索引数组。

```python
# 创建一个二维数组
x = np.random.randint(10, size=(3, 4))
print('数据数组: ', x.ndim, x.shape)
print(x)

# 使用单索引
print(x[1, 2])

# 创建x.ndim=2个一维索引数组
axis0 = np.array([0, 1, 2, 1])
axis1 = np.array([2, 1, 1, 1])

# 将一维数组分别作为数据数组x的索引，并使用`,`相隔。
new_x = x[axis0, axis1]

# 结果数组为一维，它的shape为索引数组的shape
print('数据数组: ', new_x.ndim, new_x.shape)
```

```
18  print(new_x)
```

代码解析：

- 二维数组 `x` 的 `.ndim` 为2。如果使用单索引来获得数组的元素，就需要给定数组的二维索引值(坐标)，这样才能唯一的定位到指定的元素。
- 类似于单索引，花式索引也需要指定两个 `.shape` 一致的索引数组，作为花式索引的二维索引。
- 索引数组 `axis0 = np.array([0, 1, 2, 0])` 为第1维提供索引值， `axis1 = np.array([2, 1, 1, 1])` 为第2维提供索引值。
- `new_x = x[axis0, axis1]` ，将索引数组 `axis0` 和 `axis1` 作为两个维度的索引值，来获得结果数组 `new_x` 的元素。例如， `new_x` 的第1个元素为二维索引 `(axis0[0],axis1[0])` 对应的元素： `x[0,2]` 。
- 根据规则，结果数组的 `.shape` 等于索引数组的 `.shape` 。这里的索引数组 `axis0` 和 `axis1` 为一维数组，那么结果数组也是一维数组。