第二章 Numpy计算科学模块（一）

2.1 什么是Numpy

任务：NumPy是什么?

本章和第 3 章将介绍通过 Python 有效导入、存储和操作内存数据的主要技巧。这个主题非常广泛，因为数据集的来源与格式都十分丰富，比如文档集合、图像集合、声音片段集合、数值数据集合，等等。这些数据虽然存在明显的异构性，但是将所有数据简单地看作数字数组非常有助于理解和处理数据。

> 1. 图像（尤其是数字图像）简单地看作二维数字数组，这些数字数组代表各区域的像素值；
> 2. 声音片段可以看作时间和强度的一维数组；
> 3. 文本也可以通过各种方式转换成数值表示，一种可能的转换是用二进制数表示特定单词或单词对出现的频率。

不管数据是何种形式，第一步都是**将这些数据转换成数值数组形式的可分析数据**。正因如此，有效地存储和操作数值数组是数据科学中绝对的基础过程。本课程将介绍 Python 中专门用来处理这些数值数组的工具：NumPy 包和 Pandas 包（将在第 3 章介绍）。本章将详细介绍 NumPy。

> 知识点：什么是NumPy?
>
> **NumPy是**使用Python进行科学计算的基础软件包****。在某些方面，NumPy 数组与 Python 内置的列表类型非常相似。但是随着数组在维度上变大，NumPy 数组提供了更加高效的存储和数据操作。NumPy 数组几乎是整个 Python 数据科学工具生态系统的核心。除其他外，它包括：
>
> - 功能强大的**N维数组对象**。
> - 精密**广播功能**函数。
> - 集成 C/C+和Fortran 代码的工具。
> - **强大的线性代数、傅立叶变换和随机数功能。**

> 知识点：NumPy核心技术之一：Ndarray。
>
> - NumPy 最重要的一个特点是其 **N 维数组对象 ndarray**，它是一系列同类型数据的集合，以 0 下标为开始进行集合中元素的索引。
> - ndarray 对象是用于存放同类型元素的多维数组。
> - ndarray 中的每个元素在内存中都有相同存储大小的区域。

> 知识点：NumPy核心技术之一：切片和索引。
>
> - ndarray对象的内容可以通过**索引或切片**来访问和修改，与 Python 中 list 的切片操作一样。
> - ndarray 数组可以基于 0~n 的下标进行索引，切片对象可以通过内置的 slice 函数，并设置 start，stop 及 step 参数进行，从原数组中切割出一个新数组。
> -

任务：NumPy的导包与版本号

Anaconda会自带Numpy库。安装好后，你可以导入 NumPy 并再次核实你的 NumPy 版本：

```
import numpy
numpy.__version__
```

遵循数据分析社区的传统，使用 `np` 作为别名导入 NumPy：

```
import numpy as np
np.__version__
```

任务：使用?查阅帮助文档

对内置文档的使用：Python 提供了快速探索包的内容的方法，以及各种函数的文档（用 `?` 符号）。

例如，要显示 Numpy 命名空间的所有内容，可以用如下方式：

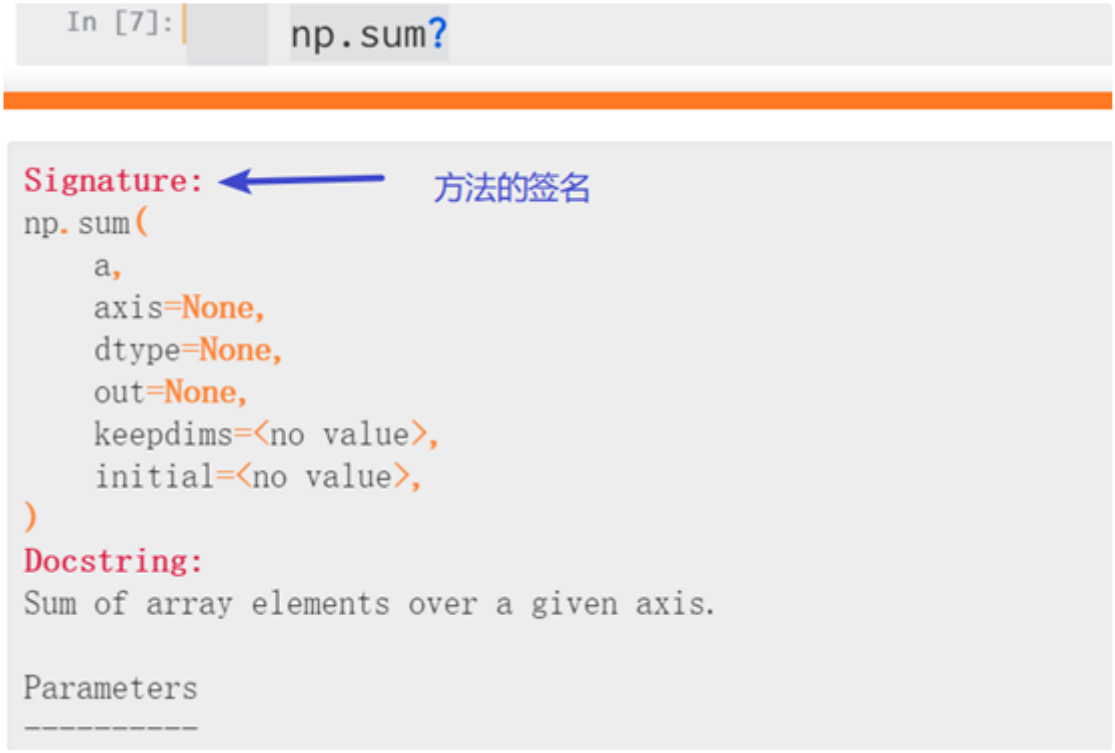
```
np.
```

当输入 `.` 号，**智能补全插件**就会给出提示。例如输入下面的代码，当输完时会提示参数输入，然后点画圈之处，可以打开更详细的方法说明(英语版)。

```
np.sum()
```

另外种方式使用方法名加 `?` 形式打开帮助文档(**不要括号**)：

```
np.sum?
```



2.2 从Python到Numpy

2.2.1 Python数据类型的缺陷

要实现高效的数据驱动科学和计算，需要理解数据是如何被存储和操作的。本节将介绍在 Python 语言中数据数组是如何被处理的，并对比 NumPy 所做的改进。理解这个不同之处是理解本节其他内容的基础。

注意：*Python*代码可以用*Jupyter*检验；*C*语言代码不能用*Jupyter*检验，需要用*C*编译器，可以跳过，仅仅用于*Python*机制的理解。

任务：掌握Python为动态类型语言，与C/Java的本质区别

> 知识点：本节将深入解释为何Python能实现动态类型机制。本质上，Python变量存的是地址，指向具体数据/对象的地址。

Python 的用户往往被其易用性所吸引，其中一个易用之处就在于**Python为动态类型语言**。静态类型的语言（如 C 或 Java）往往需要每一个变量都明确地声明，而动态类型的语言（例如 Python）可以跳过这个特殊规定。例如在 C 语言中，你可能会按照如下方式指定一个特殊的操作：

```c
/* C代码 */
int result = 0;
for(int i=0; i<100; i++){
    result += i;
}
```

而在 Python 中，同等的操作可以按照如下方式实现：

```python
# Python代码
result = 0
for i in range(100):
    result += i
```

> 注意**最大不同之处**：在 C 语言中，每个变量的数据类型被明确地声明；而在 Python 中，**类型是动态推断的。**

这意味着可以将任何类型的数据指定给任何变量：

```python
# Python代码
x = 4
x = "four"
```

这里已经将 `x` 变量的内容由整型转变成了字符串，而同样的操作在 C 语言中将会导致（取决于编译器设置）编译错误或其他未知的后果：

```c
/* C代码 */
int x = 4;
x = "four";  // 编译失败
```

这种灵活性是使 Python 和其他动态类型的语言更易用的原因之一。**理解这一特性如何工作是学习用 Python 有效且高效地分析数据的重要因素。**

但是这种类型灵活性也指出了一个事实：Python 变量不仅是它们的值，还包括了关于值的类型的一些额外信息，本节接下来的内容将进行更详细的介绍。

任务：掌握Python数据的内存存储机制

在Python中整形不仅仅是一个简单的数。下面涉及到的源码部分不需要记忆，着重理解Python数据类型的内部机制。

标准的 Python 实现是用 C 语言编写的。这意味着每一个 Python 对象都是一个聪明的伪 C 语言结构体，该结构体不仅包含其值，还有其他信息。

> 当我们在 Python 中定义一个整型，例如 `x = 10000` 时，`x` 并不是一个“原生”整型，而是一个**指针**，指向一个 C 语言的复合结构体，结构体里包含了一些值。
>
> **Python变量中存的是指向具体数据的地址。**

查看 Python 源代码，可以发现整型（长整型）的定义，如下所示（C 语言的宏经过扩展之后）：

```c
 /* C代码 */
struct _longobject {
    long ob_refcnt;
    PyTypeObject *ob_type;
    size_t ob_size;
    long ob_digit[1];
};
```

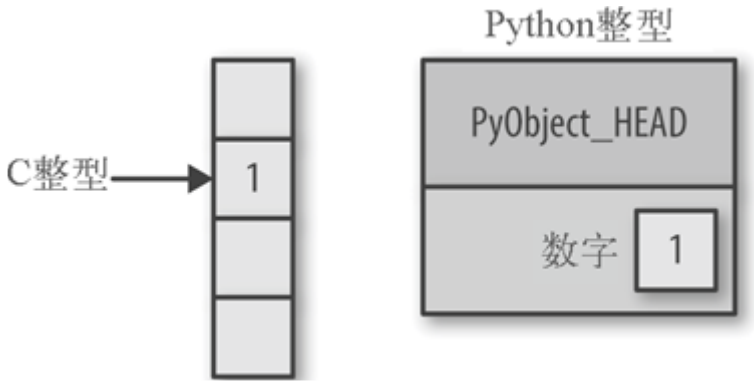
Python 中的一个整型实际上包括 4 个部分。

- `ob_refcnt` 是一个引用计数，它帮助 Python 默默地处理内存的分配和回收。
- `ob_type` 将变量的类型编码。
- `ob_size` 指定接下来的数据成员的大小。
- `ob_digit` 包含我们希望 Python 变量表示的实际整型值。

这意味着与 C 语言这样的编译语言中的整型相比，在 Python 中存储一个整型会有一些开销，下图给出了**C 整型和 Python 整型的区别**。

这里 `PyObject_HEAD` 是结构体中包含引用计数、类型编码等**辅助存储的额外信息(深入不需要理解，只需要了解Python每个数据都拥有一个额外的头)**。

- C 语言**变量直接存整形的原始数据**。比如 `int a = 1`，那么该整形数据为 `1`。在C中，**不同类型的数据，对应的存储空间也是会不一样。**所以C需要**提前声明变量的类型**，系统需要为变量分配空间。这种方式叫静态类型语言。
- Python **变量是一个指针**，指向包含这个 Python 对象所有信息的某个内存位置。该位置包括 `PyObject_HEAD` 和整形数据。任何语言的指针大小都是一样的，而指针所指的具体数据内容可以有不同的数据类型。**因此，在Python中，声明变量不需要明确类型，因为具体类型是由指针所指的具体数据所决定的，和指针本身无关。**这种方式叫动态类型语言。具体可以参加后面的补充视频资料。
- 补充：指针或地址对同一个系统来说，大小都一样。比如32位系统，指针就是32位的，这也就是为何32位系统最大支持内存；64位系统，指针就是64位的。



- **由于 Python 的整型结构体里面还包含了大量额外的信息，所以 Python 可以自由、动态地编码。**
- **Python 类型中的这些额外信息也会成为负担，在多个对象组合的结构体中尤其明显。**

2.2.2 理解Python列表内存管理机制

任务：理解Python列表内存管理机制

如果使用一个包含很多 Python 对象的 Python 数据结构(比如列表)，会发生什么?

Python 中的标准可变多元素容器是**列表**。可以用如下方式创建一个**整型值列表**：

```
L = list(range(10))
print(L)
print(type(L[0]))
```

或者创建一个**字符串列表**：

```
L2 = [str(c) for c in L]
print(L2)
print(type(L2[0]))
```

因为 Python 的动态类型特性，甚至可以创建一个**异构类型(元素含不同类型)的列表**：

```
L3 = [True, "2", 3.0, 4]
print([type(item) for item in L3])
```

但是想拥有这种**类型灵活性**也是要付出一定代价的：**为了获得这些灵活的类型，列表中的每一项必须包含各自的类型信息、引用计数和其他信息**；也就是说，每一项都是一个完整的 Python 对象。

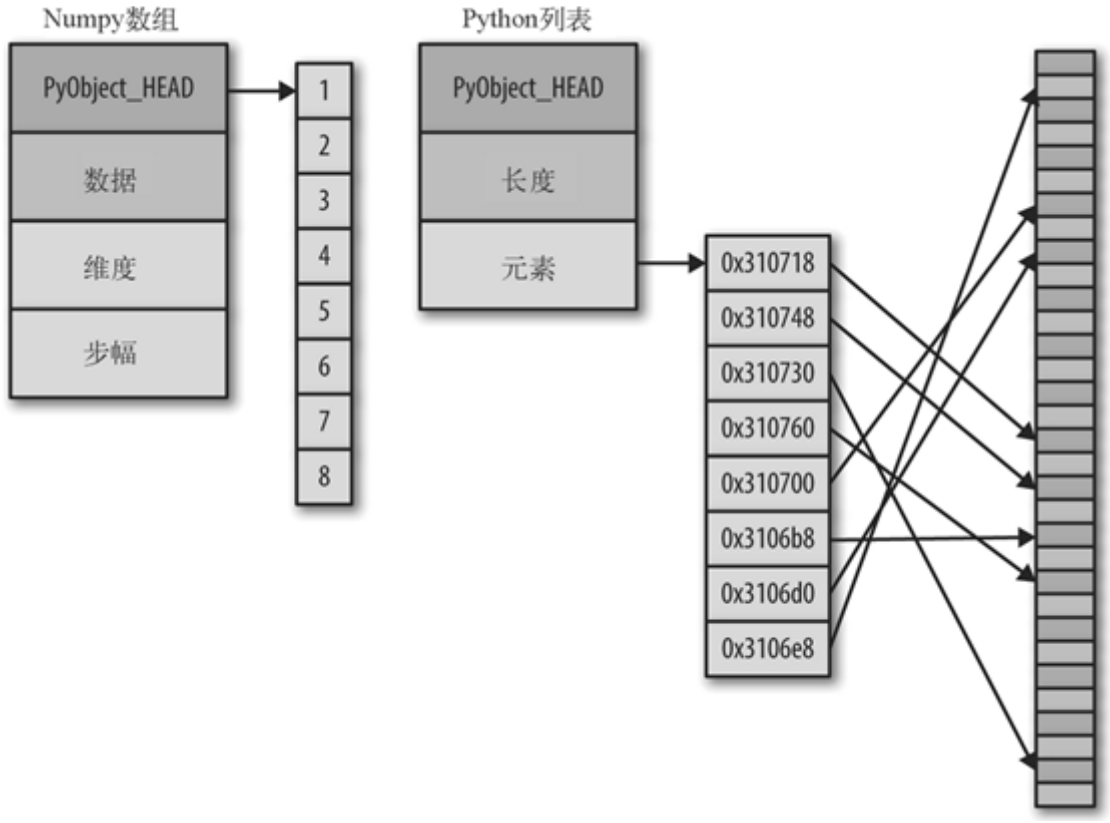
> 知识点：**列表的变量动态管理机制**，列表中每个元素存的是指向其对应真实数据的内存地址。
>
> 例如，要访问L3的第0个元素，首先通过 `L3[0]` 获取数据对应的内存地址(门牌号)；然后，根据 `L3[0]` 的内存地址间接访问 `True` 所对应的真实数据(头部HEAD和自身的数据 `True`)。
>
> 注意：**列表中存的是指向数据的地址(每个元素占用的内存大小一样)，而该地址所对应的内存块(真实值)是不一样的，内存大小也不一样。**

任务：理解Python列表存在什么不足?

> 知识点：如果列表中的**所有变量都是同一类型的，那么很多信息都会显得冗余—将数据存储在固定类型的数组中应该会更高效。**

来看一个特殊的例子，动态类型的列表和固定类型的（NumPy 式）数组间的区别如下图所示。

- 在实现层面，**Numpy数组基本上包含一个指向连续数据块的指针**。固定类型的NumPy式数组缺乏这种灵活性，但是能更有效地存储和操作数据。
- Python列表包含一个指向指针块的指针，这其中的每一个指针对应一个完整的 Python 对象（如前面看到的 Python 整型）。Python列表的优势是灵活，因为每个列表元素是一个包含数据和类型信息的完整结构体，而且列表可以用任意类型的数据填充。



- NumPy式数组：**固定类型，高效**，但是不灵活。
- Python列表：**可变类型，冗余，低效**，即可以存放各种类型数据，可充分利用碎片空间，分配数据。但由于需要额外的头部，将会造成额外的内存消耗；如果，列表元素同类型，将会造成信息冗余。

2.2.3 从Python列表到Numpy数组

创建Numpy数组有3种常规机制：

1. 从其他Python结构（例如，列表，元组）转换。
2. numpy原生数组的创建（例如，arange、ones、zeros等）。
3. 使用特殊库函数（例如，random）。

任务：创建Python中的固定类型数组

> 知识点：Python 提供了几种将数据存储在有效的、固定类型的数据缓存中的选项。内置的数组（ `array` ）模块可以用于创建统一类型的密集数组。

```
import array #导入array数组模块
L = list(range(10)) #使用列表创建 整形列表
A = array.array('i', L) #将整形列表 转化为array固定类型数组
print(A)
```

这里的 `'i'` 是一个数据类型码，表示数据为整型。

任务：Numpy模块的导包操作

> 知识点：NumPy 模块中的 `ndarray` 对象提供了高效的固定类型数组操作。

每次使用Numpy时，使用下面的语句进行导包。

```
import numpy as np
```

在同个Jupyter程序导包操作只需要运行一次(除非程序重启)，建议最开始的cell进行导包。

任务：Numpy数组创建方式一，从Python列表创建数组

> 注意：使用 `np.array` 必须要先对Numpy进行导包。否则将会出错。

首先，可以用 `np.array` 从 Python 列表创建数组：

```
# 整型数组：
np_array = np.array([1, 4, 2, 5, 3])
print(np_array, type(np_array), np_array.dtype)
```

> 知识点：不同于 Python 列表，NumPy 要求数组必须包含同一类型的数据。**如果类型不匹配，NumPy 将会向上转换**（如果可行）。

这里**整型被转换为浮点型**：

```
np_array = np.array([3.14, 4, 2, 3])
print(np_array, type(np_array), np_array.dtype)
```

> 知识点：手动设置数组的数据类型，可以用 `dtype` 关键字：

```
# 转化为浮点数
np_array = np.array([1, 2, 3, 4], dtype='float32')
print(np_array, type(np_array), np_array.dtype)
```

> 知识点：NumPy 数组可以直接创建多维数组，而Python列表只能是一维的(只能通过嵌套扩展为多维数组)。

```
# 嵌套列表构成的多维数组
np_array = np.array([range(i, i + 3) for i in [2, 4, 6]])
print(np_array, type(np_array), np_array.dtype)
```

任务：Numpy模块的随机数子模块使用

> 知识点： `np.random.random()` ： 返回规模为size的**服从均匀分布的随机浮点数数组**，位于半开区间 `[0.0, 1.0)`。

- 第一个 `random` 是模块，用于生成随机数， `np.random` 表示调用numpy中的模块 `random` 。
- 第二个 `random` 是模块 `random` 中的一个随机函数：返回服从均匀分布的随机浮点数，在半开区间 `[0.0, 1.0)`。
- 参数size：指明生成随机数组的规模大小。默认值为1，即随机生成1个标量数；如果生成二维数组，那么需要为size传入元组，例如，那么size为(n,m)，n为行数，m为列数；同理，可以生成多维数组，需要对size传入(n,m,k,...)。
- 每个Numpy数组的生成方法都拥有size参数，时候方式和这类似，下面不一样介绍。

```
res = np.random.random() #默认生成一个标量
print('标量:',res,'\n')

#(1,)代表长度为1的向量，仔细观察与默认值的区别。
res = np.random.random((1,))
print('(1,)向量:',res,'\n')

#size参数的左边值不能缺省。否则会出错
#res = np.random.random((,1))
#print(res)

#(2,3)代表规模为2*3的二阶矩阵。显示为两个[]嵌套形式。
res = np.random.random((2,3))
print('(2,3)二阶矩阵:',res,'\n')

#(2,3)代表规模为2*3*2的三阶矩阵。显示为三个[]嵌套形式。
res = np.random.random((2,3,2))
print('(2,3,2)三阶矩阵:',res,'\n')
```

> 知识点： `np.random.rand(d0, d1, ..., dn)` 函数和 `np.random.random(size)` 功能类似，返回规模为的**服从均匀分布的随机浮点数数组**，位于半开区间 `[0.0, 1.0)`。

- rand()的参数：为可变长度参数，指定每个维度的大小。
- random()的参数：size通过元组形式传递参数。

```
# 使用random和rand创建2*2规模的随机数组
# 服从[0.0 1.0)区间的均匀分布
res = np.random.random( (2,2) )
print(res)

res = np.random.rand(2,2)
print(res)
```

> 知识点： `np.random.randint(low, high, size)` :返回规模为size的**服从均匀分布的随机整形数组**，位于半开区间 `[low, high)`，不包括high这个整数。

```
np.random.randint(0, 4, (3, 3))
```

> 知识点： `np.random.normal(mean, std, size)` :返回规模为size的**服从高斯分布的随机整形数组**，mean为均值，std为方差。

```
np.random.normal(0, 2, (3, 3))
```

习题：

1. 创建一个3×3的、在0~1均匀分布的随机数组成的数组。

   代码：

   ```
   np.random.rand(3,3)
   ```

   输出结果：

   ```
   array([[0.71106764, 0.59220888, 0.28595603],
          [0.81048316, 0.8928609 , 0.61475356],
          [0.58955039, 0.64130966, 0.56994749]])
   ```

2. 创建一个3×3的、均值为0、方差为3的正态分布的随机数数组。

   代码：

   ```
   np.random.normal(0, 3, (3, 3))
   ```

   输出结果：

   ```
   array([[-1.75864783,  0.59084168,  2.47988798],
          [-0.71856686,  2.24995827, -1.94848499],
          [-2.65179068, -4.18361099, -0.23315823]])
   ```

3. 创建一个3×3的、[0, 10)区间的随机整型数组。

   代码：

   ```
   np.random.randint(0, 10, (3, 3))
   ```

   输出结果：

   ```
   array([[1, 7, 6],
          [7, 7, 5],
          [7, 2, 7]])
   ```

任务：Numpy模块的标准矩阵生成子模块使用

面对大型数组的时候，用 NumPy 内置的方法从头创建数组是一种更高效的方法。

> 知识点：
>
> - `np.zeros(size)` :返回规模为size的**全0数组**。
> - `np.ones(size)` :返回规模为size的**全1数组**。
> - `np.empty(size)` :返回规模为size的**未初始化的数组**（即数组的值保持着原内存中的数值）。
> - `np.eye(n)` :返回规模为n*n的**单位矩阵**。
> - `np.full(size,fill_value)` :返回规模为size的**用fill_value填充的数组**。
> - 这些方法还有一个可选 `dtype` 参数，指定数据类型。

```python
# 创建一个长度为10的数组，数组的值都是0
res = np.zeros(10, dtype=int)
print(res)

# 创建一个3×5的浮点型数组，数组的值都是1
res = np.ones((3, 5), dtype=float)
print(res)

# 创建一个由3个未初始化的数组，数组的值是内存空间中的任意值
res = np.empty(3)
print(res)

# 创建一个3×2的浮点型数组，数组的值都是3.14
res = np.full((3, 2), 3.14)
print(res)

# 创建一个3×1的浮点型数组，数组的值都是4，
# 这里的size，这里使用元组缺省方式(3,)，第二个为空默认为1，等价于(3,1).
res = np.full((3,), 4)
print(res)

# 创建一个由3*3个单位矩阵
res = np.eye(3)
print(res)
```

> 知识点：
>
> - `np.arange([start, ]stop, [step, ])` :返回从start到stop，步长为step的一维向量。与python的range()函数类似。start默认为0，step默认为1。参数中包含[]代表可选参数。
> - `np.linspace(start, stop, num=50)` :返回一个从start到stop区间均匀分为num份的一维向量。num默认值为50。
> - 这些方法还有一个可选 `dtype` 参数，指定数据类型。

```python
# 创建一个3×5的浮点型数组，数组的值是一个线性序列
# 从0开始，到20结束，步长为2
# （它和内置的range()函数类似）
res = np.arange(0, 20, 2)
print(res)

# 创建一个5个元素的数组，这5个数均匀地分配到0~1
res = np.linspace(0, 1, 5)
print(res)
```

习题：

1. 创建规模为4，类型为浮点数的全0矩阵。

   代码：

   ```python
   np.zeros(4, dtype=float)
   ```

   输出结果：

   ```
   array([0., 0., 0., 0.])
   ```

2. 创建规模为(3,)，类型为整形的全1矩阵。

   代码：

   ```python
   np.ones((3, ), dtype=int)
   ```

   输出结果：

   ```
   array([1, 1, 1])
   ```

3. 创建规模为(3,2)，类型为浮点的未初始化矩阵。

   代码：

   ```python
   np.empty((3,2))
   ```

   输出结果：

   ```
   array([[0., 0.],
          [0., 0.],
          [0., 0.]])
   ```

4. 创建规模为(3,2,2)，用3填充的矩阵。

   代码：

```
np.full((3,2,2),3)
```

输出结果：

```
array([[[3, 3],
        [3, 3]],

       [[3, 3],
        [3, 3]],

       [[3, 3],
        [3, 3]]])
```

5. 创建规模为(3,3)的单位矩阵。

代码：

```
np.eye(3)
```

输出结果：

```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

6. 自定义参数，使用arange和linspace创建一维数组。比较它们之间的异同点。

代码：

```
res = np.arange(0, 10, 2)
print(res)

res = np.linspace(0, 10, 5)
print(res)
```

输出结果：

```
[0 2 4 6 8]
[ 0.   2.5  5.   7.5 10. ]
```

2.2.4 NumPy标准数据类型

任务：掌握NumPy标准数据类型参数dtype的使用

NumPy 数组包含同一类型的值，因此详细了解这些数据类型及其限制是非常重要的。因为 NumPy 是在 C 语言的基础上开发的，所以 C、Fortran 和其他类似语言的用户会比较熟悉这些数据类型。

表中列出了标准 NumPy 数据类型。请注意，当构建一个数组时，你可以对参数 `dtype` 用一个字符串参数来指定数据类型：

```
np.zeros(10, dtype='int16')
```

或者使用NumPy 对象 `np.` 对参数 `dtype` 指定特定类型：

```
np.zeros(10, dtype=np.int16)
```

可以通过 `.dtype` 属性或python的 `type()` 函数**查看np数组元素的数据类型**：

```
np_array = np.zeros(10, dtype=np.int16) # np_array为numpy数组对象
# type()查询的是np数组元素类型，而非数组本身。
print(np_array, np_array.dtype, type(np_array[0]))
```

任务：掌握数组的类型转换astype()方法

要转换数组的类型，请使用 `.astype()` 方法（首选）或类型本身作为函数。注意：**类型转换函数返回值为新的类型转换后的数组，对原数组变量的类型不影响**。如果要更新原数组的类型，需要将返回值赋值给原数组变量。

```
np_array = np.arange(3, dtype=np.uint8) # np_array为numpy数组对象
print(np_array, np_array.dtype, type(np_array[0]))

# 调用.astype() 方法(建议)
np_array = np_array.astype(np.str)
print(np_array, np_array.dtype, type(np_array[0]))

# 使用类型本身
np_array = np.int8(np_array)
print(np_array, np_array.dtype, type(np_array[0]))
```

> 知识点：Numpy有5种基本数字类型表示布尔值（bool），整数（int），无符号整数（uint）、浮点（浮点数）和复数。
>
> - 名称中带有数字的那些表示该类型的位大小（即，在内存中表示单个值需要多少位）。
> - 某些类型（例如 `int` 和 `intp` ）具有不同的位，取决于平台（例如，32位与64位计算机）。

表：NumPy标准数据类型

数据类型	描述
bool_	布尔值（真、 True 或假、 False ），用一个字节存储
int_	默认整型（类似于 C 语言中的 long ，通常情况下是 int64 或 int32 ）
intc	同 C 语言的 int 相同（通常是 int32 或 int64 ）
intp	用作索引的整型（和 C 语言的 ssize_t 相同，通常情况下是 int32 或 int64 ）
int8	字节（byte，范围从-128 到 127）
int16	整型（范围从-32768 到 32767）
int32	整型（范围从-2147483648 到 2147483647）
int64	整型（范围从-9223372036854775808 到 9223372036854775807）
uint8	无符号整型（范围从 0 到 255）
uint16	无符号整型（范围从 0 到 65535）
uint32	无符号整型（范围从 0 到 4294967295）
uint64	无符号整型（范围从 0 到 18446744073709551615）
float_	float64 的简化形式
float16	半精度浮点型：符号比特位，5 比特位指数（exponent），10 比特位尾数（mantissa）
float32	单精度浮点型：符号比特位，8 比特位指数，23 比特位尾数
float64	双精度浮点型：符号比特位，11 比特位指数，52 比特位尾数
complex_	complex128 的简化形式
complex64	复数，由两个 32 位浮点数表示
complex128	复数，由两个 64 位浮点数表示

习题：

1. 分别使用字符串形式和numpy对象形式对数组进行自定义类型。自定义规模size，建立全0、全1、随机数数组。

   代码：

```
res1 = np.zeros(10, dtype='int16')
print(res1)
print(res1.astype(np.float_))

res2 = np.ones(10, dtype='int16')
print(res2)
print(res2.astype(np.float_))

res3 = np.random.randint(1,10,(10))
print(res3)
print(res3.astype(np.float_))
```

输出结果：

```
[0 0 0 0 0 0 0 0 0 0]
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
[1 1 1 1 1 1 1 1 1 1]
[1. 1. 1. 1. 1. 1. 1. 1. 1. 1.]
[9 8 8 1 1 4 6 2 5 5]
[9. 8. 8. 1. 1. 4. 6. 2. 5. 5.]
```

2. 使用arange和linspace数组自定义类型创建一维数组。

代码：

```
res = np.arange(0, 20, 2)
print(res.astype(np.float_))

res = np.linspace(0,20, 10)
print(res.astype(np.int16))
```

输出结果：

```
[ 0.  2.  4.  6.  8. 10. 12. 14. 16. 18.]
[ 0  2  4  6  8 11 13 15 17 20]
```

3. 使用 `.dtype` 属性和 `type()` 函数查看上面建立的numpy数组对象的具体数据类型。

代码：

```
np_array = np.linspace(0,20, 10)
np_array = np.int8(np_array)
print(np_array, np_array.dtype, type(np_array[0]))
```

输出结果：

```
[ 0  2  4  6  8 11 13 15 17 20] int8 <class 'numpy.int8'>
```

2.3 NumPy数组基础操作

Python 中的数据操作几乎等同于 NumPy 数组操作，甚至优秀的Pandas工具也是构建在 NumPy 数组的基础之上的。本节将展示一些用 NumPy 数组操作获取数据或子数组，对数组进行分裂、变形和连接的例子。本节介绍的操作类型可能读起来有些枯燥，但其中也包括了本书其他例子中将用到的内容，所以要好好了解这些内容！

将介绍以下几类基本的数组操作。

数组的属性：确定数组的大小、形状、存储大小、数据类型。

数组的索引：获取和设置数组各个元素的值。

数组的切片：在大的数组中获取或设置更小的子数组。

数组的变形：改变给定数组的形状。

数组的拼接和分裂：将多个数组合并为一个，以及将一个数组分裂成多个。

2.3.1 随机数生成机制

任务：理解随机数生成机制(伪随机机制)

通过设置 NumPy 的随机数生成器的**种子值**参数，以确保每次程序执行时**都可以生成同样的随机数组**。

> 知识点：什么叫 seed 随机种子？什么叫伪随机？计算机是如何控制随机序列生成的？
>
> - seed 随机种子：**计算机生成随机数序列的参数。**相同的随机种子seed，计算机会生成相同的随机序列。
> - 计算机生成的随机序列称之为**伪随机。**因为随机序列的生成依赖于 seed 种子参数；当使用相同 seed 时，生成的序列也是完全一样的。这种随机序列生成方式，**并非是完全随机的**，因此被称作**伪随机。**
> - 通过配置不同seed值，产生不同的随机数序列。 seed 参数的默认值为当前时钟，由于任何时刻的时钟是唯一的，因此默认seed每次生成的随机序列也是唯一的。

> 知识点：
>
> - np.random.seed() 函数： 配置random模块的随机种子。然后，随机序列通过 randint() 、random() 、 normal() 函数生成。
> - 当传入 seed() 相同参数时，则每次生成的随机序列会相同。如果不设置 seed ，默认为将当前时刻做为 seed ，那么此时的随机序列将会与之前的序列永远不一样，因为**当前时刻是唯一的。**

> 知识点：
>
> - 如果程序需要使用随机数函数，建议在导包之后通过 np.random.seed() 函数设置随机数种子参数。
> - **Jupyter只需要在最开始的cell运行一次，那么导包和随机种子设置对其他cell都是有效的(同个文件中)。**

```python
import numpy as np
np.random.seed(0)  # 设置随机数种子，固定每次运行的随机结果
```

创建一维、二维、三维随机整数数组。

```python
x1 = np.random.randint(10, size=6)  # 一维数组
print(x1)

x2 = np.random.randint(10, size=(3, 4))  # 二维数组
print(x2)

x3 = np.random.randint(10, size=(3, 4, 5))  # 三维数组
print(x3)
```

习题：

1. 观察结果并回答，如果设置 `seed()` 参数为一样，那么随机结果是否发生变化。任意修改 `seed()` 参数，再观察运行结果是否变化。

   代码：

   ```python
   np.random.seed(0)

   x2 = np.random.randint(10, size=(3, 4))  # 二维数组
   print(x2)

   np.random.seed(10)

   x2 = np.random.randint(10, size=(3, 4))  # 二维数组
   print(x2)
   ```

   输出结果：变了

   ```
   [[5 0 3 3]
    [7 9 3 5]
    [2 4 7 6]]
   [[9 4 0 1]
    [9 0 1 8]
    [9 0 8 6]]
   ```

2. 使用 `seed()` 参数的默认值，多次运行程序，观察结果是否变化。

   代码：

```
np.random.seed(1)

x2 = np.random.randint(10, size=(3, 4))  # 二维数组
print(x2)
```

输出结果：不变

```
np.random.seed(1)

x2 = np.random.randint(10, size=(3, 4))  # 二维数组
print(x2)
```

3．理解什么叫**伪随机数**概念。请问如何做到每次随机结果不一样？

2.3.2 NumPy数组的常见属性

任务：掌握NumPy数组的常见属性

首先定义三个随机的数组：一个一维数组、一个二维数组和一个三维数组。

```
import numpy as np
np.random.seed(0)  # 设置随机数种子

x1 = np.random.randint(10, size=6)  # 一维数组
x2 = np.random.randint(10, size=(3, 4))  # 二维数组
x3 = np.random.randint(10, size=(3, 4, 5))  # 三维数组
```

> 知识点：每个数组有 `ndim` （数组的维度）、 `shape` （数组每个维度的大小）和 `size` （数组元素数量）属性。

```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

> 知识点：数组元素的数据类型： `dtype` 属性。

```
print("dtype:", x3.dtype)
```

> 知识点：数组元素字节大小： `itemsize` 属性。数组总字节大小： `nbytes` 属性。一般来说，可以认为 `nbytes` 跟 `itemsize` 和 `size` 的乘积大小相等。

```
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

习题：

1. 随机生成一个三维数组 `size` 为 `(n,m,k)` 的数组，然后使用属性输出数据的维度、每个维度的大小、数组元素数量、数据类型。

   代码：

```
np.random.seed(1)

x3 = np.random.randint(10, size=(2, 3, 4))

print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("dtype:", x3.dtype)
```

   输出结果：

```
x3 ndim:  3
x3 shape: (2, 3, 4)
x3 size:  24
dtype: int32
```

2. 观察，数组的总字节数是否等于元素字节数*数组元素数量？

   代码：

```
np.random.seed(1)

x3 = np.random.randint(10, size=(2, 3, 4))

print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

   输出结果：不等于

```
itemsize: 4 bytes
nbytes: 96 bytes
```

2.3.3 索引访问数组元素

任务：索引访问数组元素

> 知识点：通过**中括号**指定索引获取第 个值（从 0 开始计数）。如果你熟悉 Python 的标准列表索引，那么对 NumPy 的索引方式也不会陌生。**对于一维数组，它的使用方式和Python列表一模一样。**

```python
import numpy as np
np.random.seed(0)  # 设置随机数种子
x1 = np.random.randint(10, size=6)  # 一维数组
print(x1)
print(x1[0])
print(x1[4])
```

> 知识点：使用**负索引**，访问数组的末尾进行**逆向索引**。

```python
print(x1[-1])
print(x1[-2])
```

> 知识点：Numpy相对于Python列表，**直接支持多维数组**(一个重大的技术突破)。Numpy数据可以通过索引，直接访问多维数组的元素。**多维数组中，可以用逗号分隔的索引元组获取元素。** Python列表并没有实现多维列表功能，虽然它可以使用嵌套表示多维列表，但是元素索引访问比较繁琐。

```python
x2 = np.random.randint(10, size=(3, 4))  # 二维数组
print(x2)
print(x2[0, 0])
print(x2[2, 0])
print(x2[2, -1])
```

> 知识点：类似于Python，可以**修改索引访问的元素的值**。

```python
x2[0, 0] = 12
print(x2)
```

> 知识点：和 Python 列表不同，**NumPy 数组是固定类型**的。这意味着当试图将一个浮点值插入一个整型数组时，**浮点值会被自动截短成整型**。并且这种截短是自动完成的，**不会给你提示或警告**，所以需要特别注意这一点！

```python
x2[0, 0] = 3.14159  # 这将被截断
print(x2)
```

习题：

1. 随机生成一个三维数组 `size` 为 `(n,m,k)` 的**整数**数组，使用正索引访问元素，负索引访问元素，正负索引组合访问元素。

   代码：

```
np.random.seed(1)

x3 = np.random.randint(10, size=(2, 3, 4))
print(x3)

print(x3[0,0,1])
print(x3[-1,-1,-1])
print(x3[0,0,-1])
```

输出结果：

```
[[[5 8 9 5]
  [0 0 1 7]
  [6 9 2 4]]

 [[5 2 4 2]
  [4 7 7 9]
  [1 7 0 6]]]
8
6
5
```

2. 对索引元素修改值，观察结果；如果修改的值为浮点数，那么结果又是什么？请分别讨论。

   浮点值会被自动截短成整型

第三章 Numpy科学计算模块（二）

3.1 numpy数组的切片

3.1.1 切片的定义

任务：切片的定义

> 知识点： 类似于Python的列表，Numpy数组同样可以使用**切片**(slice)来获取子数组，切片操作符用冒号 `:` 表示。NumPy切片语法和Python列表的标准切片语法相同。

语法：为了获取数组 `x` 的一个切片(子数组)，使用：

```
x[start:stop:step]
```

- 一个完整的切片表达式包含两个 `:` 。
- `start` 为开始索引； `stop` 为终止索引(**不包括当前索引元素**)； `step` 为步长。
- 当只有一个 `:` 时，默认第三个参数 `step=1` ；
- 当都没有 `:` 时， `start=end` ，表示切取 `start` 指定的那个元素。此时，退化为**单索引**。
- 如果以上 3 个参数都未指定，那么它们会被分别设置默认值 `start=0` 、 `stop=维度的大小` (`size of dimension`)和 `step=1` 。此时， `x(::)` 为全数组 `x` 本身。

3.1.2 一维数组切片的使用

任务：一维数组切片的使用

> 知识点：一维数组的切片和list切片的使用类似。**注意**：切片获得的子数组不包括 `stop` 索引的元素，子数组的范围是从 `start` 到 `stop-1` 。

```
x = np.arange(10)
print(x)
print(x[:5])  # 前五个元素
print(x[5:])  # 索引五之后的元素
print(x[4:7])  # 中间的子数组
print(x[::2])  # 每隔一个元素
print(x[1::2])  # 每隔一个元素，从索引1开始
```

> 知识点：当 `step` 为负值时， `start` 为最大索引元素、 `stop` 为最小索引元素(不包括此元素)。此时，输出逆序子数组。逆序子数组的范围是从 `start` 到 `stop-1` 。

```
x = np.arange(10)
print(x)
print(x[::-1]) # 所有元素，输出逆序全数组。
print(x[6:2:-1]) #从索引6开始，到索引2结束(不包括)，逆序子数组。
print(x[5::-2]) # 从索引5开始，每隔一个元素，输出逆序子数组。
```

习题：

1. 随机生成一个 `size=(10,)` 的一维数组。通过切片操作，获得 `[2,7)` 的子数组；获得 `[8,1)` 的逆序子数组；获得 `[4,10]` 间隔为2的子数组；获得 `[6,1]` 间隔为3的逆序子数组。

   代码：

   ```
   x = np.arange(10,)

   print(x)
   print(x[2:7])
   print(x[8:1:-1])
   print(x[4:10:2])
   print(x[6:2:-3])
   ```

   输出结果：

   ```
   [0 1 2 3 4 5 6 7 8 9]
   [2 3 4 5 6]
   [8 7 6 5 4 3 2]
   [4 6 8]
   [6 3]
   ```

3.1.3 多维数组切片的使用

任务：多维数组切片的使用

> 知识点：多维数组与一维数组类似，分别对每个维度的索引进行切片操作。
>
> - 多维通过 `,` 相隔，比如对二维数组的切片： `x[start:stop:step, start:stop:step]` 。
> - 理解：**多维索引可以类比为多维空间的坐标。 那么多维切片，就是获得元素的多维索引坐标。**

> 知识点：
>
> - 对于二维数组，通常将第1维称为行，第2维称为列。
> - 对于多维数组，一般直接称某一维，比如第1维，第3维。

```
import numpy as np
np.random.seed(0)  # 设置随机数种子

x2 = np.random.randint(10, size=(3, 4)) #生成随机整数数组(3,4)
print(x2)
print(x2[:2, :3])
```

- `x2[:2, :3]` 切片的子数组元素，可以先按一维切片方式理解。
- 第一维： `:2` 对应切片范围为 `[,2)` ，索引为 `0,1` ，规模为2。
- 第二维： `:3` 对应切片范围为 `[,3)` 。索引为 `0,1,2` ，规模为3。
- **再次强调，多维索引可以类比为多维空间的坐标。** 组合两维的索引，可得二维子数组的规模为 `size=2*3` ，2行3列。
- 使用遍历枚举方式，二维子数组的第0行元素为 `[x2[0,0], x2[0,1], x2[0,2]]` ；第1行元素为 `[x2[1,0], x2[1,1], x2[1,2]]` 。

习题：

1. 随机生成二维的整形数组 `size=(6,7)` 。为每个子数组输出 `ndim` 和 `shape` 。
2.
   - 取第3行的所有元素；
   - 取第4列的所有元素；
   - 取第3行，第 `[0,2)` 范围的列元素；
   - 取第2列，第 `[0,4)` 范围的行元素；
   - 取第 `[0,2)` 范围的行和第 `[2,5)` 范围的列元素；
   - 行按间隔为2和列按间隔为3；
   - 逆序操作数组。
3. 代码：

4.
```
np.random.seed(0)
x2 = np.random.randint(10, size=(6, 7))

print(x2)
print(x2[2:3,:7])
print(x2[3:4,:7])
print(x2[2:3,:2])
print(x2[:4,1:2])
print(x2[:2,2:5])
print(x2[::2,::3])
print(x2[::-1,::-1])
```

5. 输出结果：

6.
```
[[5 0 3 3 7 9 3]
 [5 2 4 7 6 8 8]
 [1 6 7 7 8 1 5]
 [9 8 9 4 3 0 3]
 [5 0 2 3 8 1 3]
 [3 3 7 0 1 9 9]]
[[1 6 7 7 8 1 5]]
[[9 8 9 4 3 0 3]]
[[1 6]]
[[0]
 [2]
 [6]
 [8]]
[[3 3 7]
 [4 7 6]]
[[5 3 3]
 [1 7 5]
 [5 3 3]]
[[9 9 1 0 7 3 3]
 [3 1 8 3 2 0 5]
 [3 0 3 4 9 8 9]
 [5 1 8 7 7 6 1]
 [8 8 6 7 4 2 5]
 [3 9 7 3 3 0 5]]
```

3.1.4 多维数组的单索引切片的使用

任务：多维数组的单索引切片的使用

> 知识点：对于 `N` 维数组，通过对某个维度进行**单索引**，获得 `N-1` 维的子数组。
>
> - 二维数组，获得第n行元素(一维数组)， `x[n,:]` 。第m列元素(一维数组)， `x[:,m]` 。
> - 三维数组， `x[:,n,:]` 为二维子数组，表示为在第2维度上对第n个索引进行切片(其他维度规模不变)。

- 理解：三维x-y-z空间中，在y轴上的第n个坐标点上切一刀(沿着平行于x和z轴方向)，获得一个二维平面。
- 再高维的话，可以抽象理解。从线性代数角度， `N` 维空间的 `N-1` 维子空间。

```
print(x2)
print(x2[:, 2]) # 切片为所有行元素，第2列为元素。
print(x2[1, :]) # 切片为第1行，所有列元素。
```

知识点：在子数组中，通过对某个维度使用 `:` 切片，保持该维度元素规模不变。二维数组 `x[:, start:stop:step]` ，第1维保持不变，只对第2维进行切片操作。

```
print(x2[:, ::2]) # 切片为所有行元素，列为间隔元素。
```

知识点：当 `step` 为负值时，获得多维子数组的逆序操作。

```
print(x2[::-1, ::-1])
```

习题：

1. 随机生成三维的整形数组 `size=(3,4,2)` 。为每个子数组输出 `ndim` 和 `shape` 。
2.
   - 在第2维上取第2个索引，其他维保持不变。 `x[:,1,:]` 。此时的子数组是几维的?
   - 在第3维上取第1个索引，其他维保持不变。此时的子数组是几维的?
   - 在第1维上取第2个索引，在第2维上取第1个索引，第3维保持不变。此时的子数组是几维的?
   - 设计5个子数组切片规则(包括逆序)，输出结果，并分析切片方式。
3. 代码：
4.
```
np.random.seed(0)
x3 = np.random.randint(10,size=(3,4,2))

print(x3[:,1,:])
print(x3[:,:,0])
print(x3[1,0,:])
```
5. 输出结果：
6.
```

```

3.1.5 获取二维数组的行和列

任务：获取二维数组的行和列

> 知识点：获取二维数组的单行和单列。可以将索引与切片组合起来实现这个功能，用一个冒号（ : ）表示空切片。

```
x2 = np.random.randint(10, size=(3, 4)) #生成随机整数数组(3,4)
print(x2[:, 0])  # x2的第一列
print(x2[0, :])  # x2的第一行
```

任务：切片 : 的省略

> 知识点：
>
> - 切片语法上，对于多维数组，如果高维的切片使用 : ，可以 : 省略。
> - 反之，不成立，如果低维的切片使用 : ，那么该 : 不可省略。
> - x 是2维数组： x[0,:] 可等价于 x[0] ；而 x[:,0] 不可等价于 x[0] 。
> - x 是3维数组： x[0,:,:] 可等价于 x[0] ； x[:,0,:,] 可等价于 x[:,0] 。

```
x2 = np.random.randint(10, size=(3, 4)) #生成随机整数数组(3,4)
print(x2[0,:])
print(x2[:,0])
print(x2[0])  #等于x2[0, :]，不等价于x2[:,0]
x3 = np.random.randint(10, size=(2, 3, 4)) #生成随机整数数组(2,3,4)
print(x3[:,2,:])
print(x3[:,2])
```

习题：

1. 随机生成一个二维数组，使用切片获得单行和单列。

   代码：

   ```
   np.random.seed(0)
   x2 = np.random.randint(10, size=(3, 3))

   print(x2[:, 0]) #列
   print(x2[0, :]) #行

   ```

   输出结果：

```
[5 3 3]
[5 0 3]
[[8 1]
 [6 7]]
[[7 8]
 [1 7]]
```

2. 随机生成一个三维数组，使用切片获得第1维第某个索引的切平面、第3维某个索引的切平面。

   代码：

```
x3 = np.random.randint(10, size=(2,2,2))

print(x3[1,:,:])
print(x3[:,:,1])

```

   输出结果：

```
[[9 8]
 [9 4]]
[[8 5]
 [8 4]]
```

3. 使用切片的 `:` 省略方式，获取子数组，并验证分析什么时候 `:` 可以使用，什么时候不可用。

4.