NUMPY-ndarray

○ 第七组　　○ 主讲人：陈诺　○ 组员：陈诺，李世昊，唐嘉铖，张钰栋

Numpy引入

Numpy

NumPy(Numerical Python) 是 Python 语言的一个扩展程序库，支持大量的维度数组与矩阵运算，此外也针对数组运算提供大量的数学函数库。它包含：

- 一个强大的N维数组对象 ndarray
- 广播功能函数
- 整合 C/C++/Fortran 代码的工具
- 线性代数、傅里叶变换、随机数生成等功能

ndarray

设计哲学

ndarray的设计哲学在于数据存储与其解释方式的分离，或者说‘副本’和‘视图’的分离，让尽可能多的操作发生在解释方式上（‘视图’上），而尽量少地操作实际存储数据的内存区域。

副本是一个数据的完整的拷贝，如果我们对副本进行修改，它不会影响到原始数据，物理内存不在同一位置。

视图是数据的一个别称或引用，通过该别称或引用亦便可访问、操作原有数据，但原有数据不会产生拷贝。如果我们对视图进行修改，它会影响到原始数据，物理内存在同一位置。

视图一般发生在：

1、numpy 的切片操作返回原数据的视图。

2、调用 ndarray 的 view() 函数产生一个视图。

副本一般发生在：

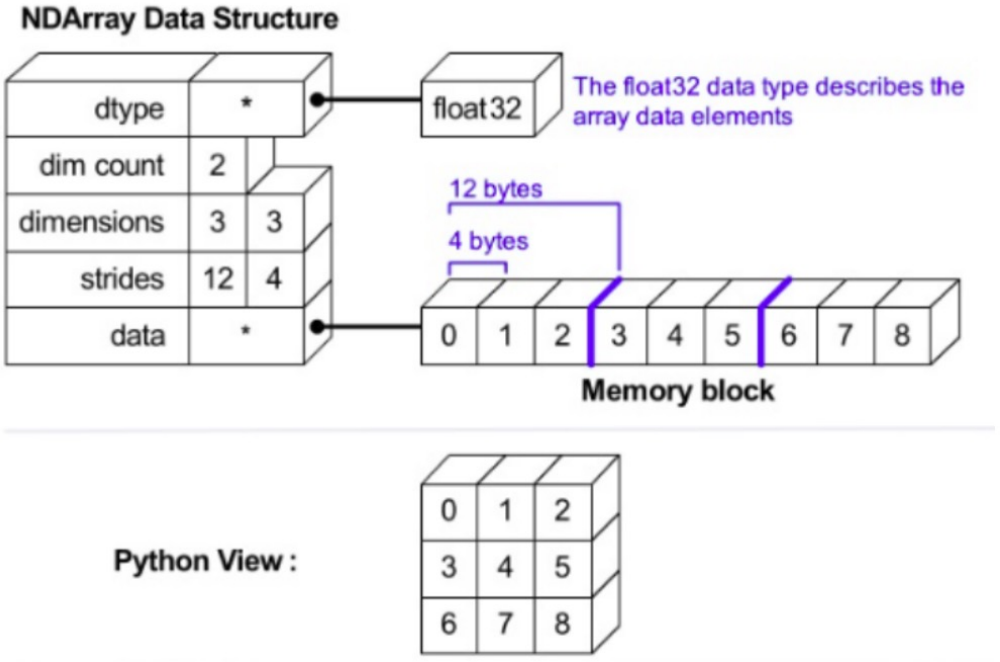
Python 序列的切片操作，调用deepCopy()函数。

调用 ndarray 的 copy() 函数产生一个副本。

ndarray

内存布局

ndarray内存布局示意图如下



内存布局

```
import numpy as np
a = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
a
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
a.dtype
```

```
dtype('int64')
```

```
a[1,2]
```

```
6
```

```
a[:,1:3]
```

```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

```
a.ndim
```

```
2
```

```
a.shape
```

```
(3, 4)
```

```
a.strides
```

```
(32, 8)
```

```
b = a.reshape(4, 3)
b
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
# reshape操作产生的是view视图，只是对数据
#的解释方式发生变化，数据物理地址相同
a.ctypes.data
```

```
140533973156400
```

```
b.ctypes.data
```

```
140533973156400
```

```
id(a) == id(b)
```

```
False
```

```
# 数据在内存中连续存储
from ctypes import string_at
string_at(b.ctypes.data, b.nbytes).hex()
```

```
'000000000000000001000000000000000200000000000000030000000000
```

```
# b的转置c，c仍共享相同的数据block，只改变了数据的解释方式
#，“以列优先的方式解释行优先的存储”
c = b.T
c
```

```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

```
c.ctypes.data
```

```
140533973156400
```

```
string_at(c.ctypes.data, c.nbytes).hex()
```

```
'000000000000000001000000000000000200000000000000030000000000
```

内存布局

```
a
```

Python

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

```
# copy会复制一份新的数据，其物理地址位于不同的区域
c = b.copy()
c
```

```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

```
c.ctypes.data
```

```
140533159951984
```

```
# slice操作产生的也是view视图，仍指向原来数据block中的物理地址
d = b[1:3, :]
d
```

```
array([[3, 4, 5],
       [6, 7, 8]])
```

```
d.ctypes.data
```

```
140533973156424
```

```
print('data buff address from {0} to {1}'.format(b.ctypes.data, b.ctypes.data + b.nbytes))
```

```
data buff address from 140533973156400 to 140533973156496
```

```
string_at(c.ctypes.data, c.nbytes).hex()
```

Python

```
'000000000000000001000000000000000200000000000000030000000000000004000000000000000500000000000000060000000000000007000000000000000800000000000000090000000000000
```

ndarray

内存布局

可大致划分成2部分——对应设计哲学中的数据部分和解释方式：
raw array data：为一个连续的memory block，存储着原始数据，类似C或Fortran中的数组，连续存储
metadata：是对上面内存块的解释方式

metadata包含信息：

dtype：数据类型，指示了每个数据占用多少个字节，这几个字节怎么解释，比如int32、float32等；

ndim：有多少维；

shape：每维上的数量；

strides：维间距，即到达当前维下一个相邻数据需要前进的字节数，因考虑内存对齐，不一定为每个数据占用字节数的整数倍；

上面4个信息构成了ndarray的indexing schema，即如何索引到指定位置的数据，以及这个数据该怎么解释。

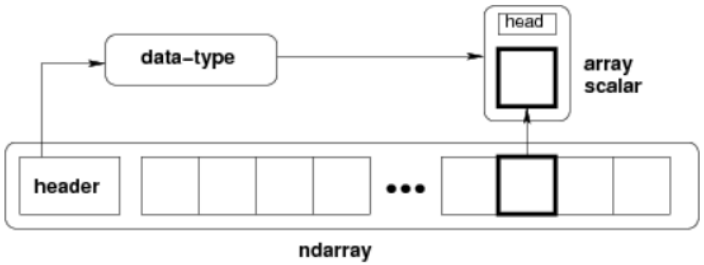
除此之外的信息还有：字节序（大端小端）、读写权限、C-order（行优先存储） or Fortran-order（列优先存储）等

ndarray

设计原因

为什么ndarray可以这样设计？

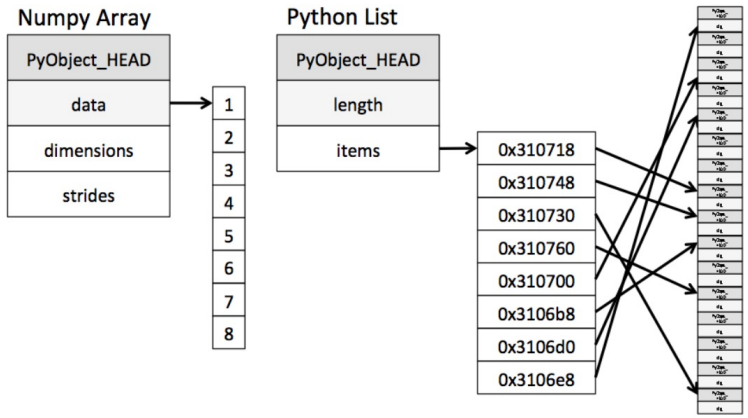
因为ndarray是为矩阵运算服务的，ndarray中的所有数据都是同一种类型，比如int32、float64等，每个数据占用的字节数相同、解释方式也相同，所以可以稠密地排列在一起，在取出时根据dtype现copy一份数据组装成scalar对象输出。这样极大地节省了空间，scalar对象中除了数据之外的域没必要重复存储，同时因为连续内存的原因，可以按秩访问，速度也要快得多。



ndarray

设计原因

这里，可以将ndarray与python中的list对比一下，list可以容纳不同类型的对象，像string、int、tuple等都可以放在一个list里，所以list中存放的是对象的引用，再通过引用找到具体的对象，这些对象所在的物理地址并不是连续的，如下所示：



所以相对ndarray，list访问到数据需要多跳转1次，list只能做到对对象引用的按秩访问，对具体的数据并不是按秩访问，所以效率上ndarray比list要快得多，空间上，因为ndarray只把数据紧密存储，而list需要把每个对象的所有域值都存下来，所以ndarray比list要更省空间。

小结

- ndarray的设计哲学在于数据与其解释方式的分离，让绝大部分多维数组操作只发生在解释方式上；
- ndarray中的数据在物理内存上连续存储，在读取时根据dtype组装成对象输出，可以按秩访问，效率高省空间；
- 之所以能这样实现，在于ndarray是为矩阵运算服务的，所有数据单元都是同种类型。

Pre

2023 年 5 月 7 日

[ ]:
```
import numpy as np
a = np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11]])
a
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
a.dtype
```

[ ]:
```
dtype('int64')
```

[ ]:
```
a[1,2]
```

[ ]:
```
6
```

[ ]:
```
a[:,1:3]
```

[ ]:
```
array([[ 1,  2],
       [ 5,  6],
       [ 9, 10]])
```

[ ]:
```
a.ndim
```

[ ]:
```
2
```

[ ]:
```
a.shape
```

[ ]:
```
(3, 4)
```

[ ]:
```
a.strides
```

[ ]:
```
(32, 8)
```

[ ]:
```
b = a.reshape(4, 3)
b
```

[ ]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

[ ]:
```
# reshape 操作产生的是 view 视图，只是对数据的解释方式发生变化，数据物理地址相同
a.ctypes.data
```

[ ]:
```
140533973156400
```

[ ]:
```
b.ctypes.data
```

[ ]:
```
140533973156400
```

[ ]:
```
id(a) == id(b)
```

[ ]:
```
False
```

[ ]:
```
# 数据在内存中连续存储
from ctypes import string_at
string_at(b.ctypes.data, b.nbytes).hex()
```

[ ]:
```
'000000000000000001000000000000000200000000000000030000000000000004000000000000000500000000000000060000000000000007000000000000000800000000000000090000000000000000a000000000000000b00000000000000'
```

[ ]:
```
# b 的转置 c，c 仍共享相同的数据 block，只改变了数据的解释方式，“以列优先的方式解释行优先的存储”
c = b.T
c
```

[ ]:
```
array([[ 0,  3,  6,  9],
       [ 1,  4,  7, 10],
       [ 2,  5,  8, 11]])
```

[ ]:
```
c.ctypes.data
```

[ ]:
```
140533973156400
```

[ ]:
```
string_at(c.ctypes.data, c.nbytes).hex()
```

[ ]:
```
'00000000000000001000000000000000200000000000000030000000000000004000000000000000
0050000000000000060000000000000070000000000000008000000000000000900000000000000
00a000000000000000b00000000000000'
```

[ ]:
```
a
```

[ ]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[ ]:
```
# copy 会复制一份新的数据，其物理地址位于不同的区域
c = b.copy()
c
```

[ ]:
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

[ ]:
```
c.ctypes.data
```

[ ]:
```
140533159951984
```

[ ]:
```
string_at(c.ctypes.data, c.nbytes).hex()
```

[ ]:
```
'00000000000000001000000000000000200000000000000030000000000000004000000000000000
0050000000000000060000000000000070000000000000008000000000000000900000000000000
00a000000000000000b00000000000000'
```

[ ]:
```
# slice 操作产生的也是 view 视图，仍指向原来数据 block 中的物理地址
d = b[1:3, :]
d
```

[ ]:
```
array([[3, 4, 5],
       [6, 7, 8]])
```

[ ]:
```
d.ctypes.data
```

[ ]:
```
140533973156424
```

[ ]:
```
print('data buff address from {0} to {1}'.format(b.ctypes.data, b.ctypes.data +␣
↪b.nbytes))
```

```
data buff address from 140533973156400 to 140533973156496
```