关于本节中要掌握的内容

- array概念与创建
- array访问与修改

概念与创建

在之前的学习中，我们接触了list，现在又要学习numpy的array，首先回答一个问题，为什么有了list，还要学习array。主要是以下几个原因：

- 更好的性能，仅在求和方面的表现来看，numpy的速度比list要快10倍，不同的机器可能会有不同的表现。
- 更小的内存，numpy使用C语言实现，每个元素的大小和类型一致，不需要额外的空间存储每个元素的信息
- 更强的功能，numpy提供了丰富的数组操作和函数，提供了对数据统计，数学运算，矢量计算等功能；提供了广播功能，能够更简单的处理不同形状的数组；提供了更丰富的索引和切片的功能

关于numpy和list，简单说明使用范围：

- 对于自动化任务，例如处理文件、简单数据、字符串等操作使用list即可。
- 对于复杂数据计算，例如矩阵、向量等数据，统计分析等功能使用numpy

创建array

```
import numpy as np
arr1 = np.array([0, 1, 2, 3, 4])
arr2 = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0]])
print(arr1)
print(arr2)
# 输出array的维度
print(arr1.ndim,arr2.ndim)
# 输出array的形状
print(arr1.shape,arr2.shape)
# 输出array的数据类型
print(arr1.dtype,arr2.dtype)
# 输出array的元素个数
print(arr1.size,arr2.size)
```

1.下面的array的ndim是多少?

```
arr3 = np.array([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0,7.0,8.0]])
print(arr3.ndim)
```

2.下面的array的dtype是什么?

```
arr4 = np.array([[0, 1, 2], [3.0, 4.0, 5.0]])
print(arr4.dtype)
```

3.arr1和arr2的每个元素大小是多少?

```
print(arr1.itemsize,arr2.itemsize)
```

4.arr1和arr2占用内存大小是多少?

```
print(arr1.nbytes,arr2.nbytes)
```

在创建np.array时，我们有以下几种方式：

```
arr1 = np.array([1, 2, 3])
arr2 = np.array((1, 2, 3))
arr3 = np.array([1.0, -2.0, 3.0], dtype=np.uint64)
arr4 = np.array([[1, 2, 3], [4, 5, 6]])
arr5 = np.array(([1, 2, 3], [4, 5, 6]))
arr6 = np.array(arr4)
arr7 = np.asarray(arr4)
```

- np.array可以使用list/元组来构造
- np.array可以自定义数据类型
- np.array复制另一个array，asarray引用另一个array

练习题

1.创建一个1维array，其中元素为4,5,6
2.创建一个1维的string array
3.创建一个2维array，其中元素为1,2,3 4.0,5.0,6.0
4.创建任意一个3维array

可能的实现

```
arr1 = np.array([4,5,6])
arr2 = np.array(['a','b','c'])
arr3 = np.array([[1,2,3],[4.0,5.0,6.0]])
arr4 = np.array([[[1,2,3],[4,5,6]],[[2,3,4],[5,6,7]]])
```

- 使用zeros创建array

```
import numpy as np
np.zeros(10)
np.zeros((2,3))
np.zeros((4,3),dtype=np.int64)
```

- 使用ones创建array

```
import numpy as np
np.ones(10)
np.ones((2,3))
np.ones((4,3),dtype=np.int64)
```

- 使用full创建array

```
import numpy as np
np.full(10,100)
np.full((2,3),100)
np.full((4,3),100,dtype=np.float32)
```

- 使用已有array创建array

```
arr1 = np.zeros((4,3),dtype=np.int64)
arr2 = np.zeros(arr1.shape)
arr3 = np.zeros_like(arr1)
arr4 = np.ones_like(arr1)
arr5 = np.full_like(arr1,10)
```

注意事项

- zeros和ones默认类型为float64，full默认类型为int64，可以通过dtype来自定义数据类型
- 可以通过zeros_like、ones_like、full_like创建一个和目标array相同形状、相同类型的array；如果仅使用shape，只能保证形状相同

练习题

1.创建一个形状为 (3, 4) 的2维数组，其中所有元素均为0（int）

2.创建一个形状为 (4, 5) 的2维数组，其中所有元素均为1（float）

3.创建一个形状为 (3, 4, 5) 的3维数组，其中所有元素均为10.0(float)

4.使用练习题2的array，创建一个和它形状相同的array，所有元素都为0（int）

5.使用练习题4的array，创建一个和它形状、类型都相同的array，所有元素都为1

可能的实现

```
arr1 = np.zeros((3,4),dtype=np.int64)
arr2 = np.ones((4,5))
arr3 = np.full((3,4,5),10.0)
arr4 = np.zeros(arr2.shape,dtype=np.int64)
arr5 = np.ones_like(arr4)
```

- 创建对角矩阵

```
import numpy as np
np.eye(3)
np.eye(3,4)
np.eye(4,5,dtype=np.int32)
np.eye(3,4,k=1)
np.eye(3,4,k=-1)
```

- 创建空数组

```
import numpy as np
np.empty((4, 3))
```

需要注意的点

- np.eye()实际创建的是2维数组，并不是真的mat
- 通常使用np.empty是为了初始化/插入的效率，需要注意的是，使用empty创建的array之前必须要初始化它

- 比较np.array和list的append的效率

```
import numpy as np
import timeit
# 测试 np.array 的性能
def test_np_array():
    arr = np.array([])  # 创建一个空的 numpy 数组
    for i in range(10000):
        arr = np.append(arr, i)
# 测试 list 的性能
def test_list():
    lst = []  # 创建一个空的 Python 列表
    for i in range(10000):
        lst.append(i)
# 测试性能
np_array_time = timeit.timeit(test_np_array, number=1000)
list_time = timeit.timeit(test_list, number=1000)
# 输出结果
print("np.array append time:", np_array_time)
print("list append time:", list_time)
```

- 比较使用empty后的效率变化

```
import numpy as np
import timeit
# 预先分配空间并填充元素
def append_to_np_array(n):
    arr = np.empty(n, dtype=np.int32)  # 预先分配 n 个元素的空间
    for i in range(n):
        arr[i] = i
    return arr
# 使用 list.append() 进行比较
def append_to_list(n):
    lst = []
    for i in range(n):
        lst.append(i)
    return np.array(lst)
# 测试性能
n = 100000
np_array_time = timeit.timeit(lambda: append_to_np_array(n), number=100)
list_time = timeit.timeit(lambda: append_to_list(n), number=100)
# 输出结果
print("np.array append time:", np_array_time)
print("list.append() time:", list_time)
```

- 这里的dtype使用np.int32要比使用np.uint32要快，为什么呢?

- 生成等差数组

```
l1 = list[i for i in range(10)]
arr1 = np.arange(10)
arr2 = np.arange(0,10)
arr3 = np.arange(0,10,1)
arr4 = np.arange(0,10,1.0)
```

- 生成随机数组

```
np.random.seed(100)
# 生成[0,1)之间的float随机数
arr1 = np.random.random(5)
arr2 = np.random.rand(3, 2)
# 生成[2,3)之间的float随机数
np.random.uniform(2, 3, 5)
np.random.uniform(2, 3, (2, 3))
# 生成[1,10)之间的int随机数
np.random.randint(1, 10, (2,3))
# 生成均值为0且标准差为1的正态分布随机数
np.random.randn(3, 4)
```

练习题

1.创建一个随机整数数组，形状为 (4, 5)，取值范围在 1 到 10 之间

2.创建一个3*4的对角矩阵，对角线偏移2

3.创建一个形状为(4,5)的空数组

4.创建一个float数组，形状为（3，4），要求为从0开始的等差为0.1的数组

可能的实现

```
arr1 = np.random.randint(1,10,(4,5))
arr2 = np.eye(3,4,2)
arr3 = np.empty((3,4))
arr = np.arange(0, 12) * 0.1
arr4 = arr.reshape((3, 4))
```

访问与修改

访问数组

为了快速访问数组中的某个/某列/某些列/某行/某些行/某些行列....的数据，numpy提供了一系列的方式，包括下标、数组切片、索引切片等....

还记得之前是如何访问list元素的吗？访问数组元素基本相似：

```
arr1 = np.arange(12)
arr1 = np.reshape(arr1, (3,4))
print(arr1)
print(arr1[0],arr1[2],arr1[-2])
print(arr1[0,1],arr1[-1,-2],arr1[1,-3])
```

- 获取数组切片

```
arr1 = np.arange(12)
# 获取arr1的前6个元素
print(arr1[:6])
# 获取arr1的后4个元素
print(arr1[8:])
# 获取arr1的第4个元素（含）到第9个元素（不含）
print(arr1[4:9])
# 获取arr1的所有奇数
print(arr1[1::2])
# 获取arr1的倒序的元素
print(arr1[::-1])
```

```
arr2 = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(arr2[:2, :2])
print(arr2[:3, ::2])
print(arr2[::-1, ::-1])
print(arr2[:2])
arr2[slice(0, 2)]
```

- 索引结合切片

```
arr2 = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
print(arr2[1, :2])
print(arr2[-2:, 0])
```

练习题

1.创建一个形状为 (3, 4) 的二维数组，命名为 arr，并访问其第二行。

2.在上述数组 arr 中，访问第三列。

3.创建一个一维数组 a，包含数字 0 到 9。然后，通过数组切片访问第二到第五个元素。

4.创建一个 3x3 的二维数组，然后使用切片访问其右下角的 2x2 子数组。

5.创建一个 4x4 的随机整数矩阵，使用切片访问其中的第二和第三行。

可能的实现

```
import numpy as np
np.random.seed(100)
arr1 = np.random.randint(1,100,(3,4))
print(arr1)
print(1,arr1[1])
print(2,arr1[:,2])
print(3,np.arange(10)[1:5])
arr2 = np.random.randint(1,100,(3,3))
print(arr2)
print(4,arr2[1:,1:])
arr3 = np.random.randint(1,100,(4,4))
print(arr3)
print(5,arr3[1:3])
```

修改数组

关于更新数组的元素，最简单就是对一个元素进行赋值了，例如：

```
arr2 = np.array([[0, 1, 2], [3, 4, 5], [6, 7, 8]])
arr2[1, 2] = 100
```

当然，除了简单的赋值外，还可以对数组切片进行赋值，例如：

```
arr2[0] += 10
arr2[1] = 25
arr2[2] = [9,10,11]
arr2[:2, :2] = 0
arr2[1:, 1:] = [[0, 1], [2, 3]]
arr2[-1, ::2] = 5
arr2[arr2<5] = 1
```

练习题

1.创建一个形状为 (3, 3) 的二维数组，将其中一行的所有元素都设置为相同的值，例如 1。

2.创建一个一维数组，将数组的前三个元素设置为零。

3.创建一个 4x4 的随机整数矩阵，将矩阵的第二列所有元素都加上 10。

4.创建一个形状为 (3, 3) 的零矩阵，然后使用循环将矩阵的每个元素设置为该元素在矩阵中的行索引。

5.创建一个包含 10 个随机整数的一维数组，将其中的偶数元素都乘以 2。

6.创建一个 3x3 的矩阵，将其中所有小于 5 的元素都替换为 5。

7.创建一个 2x2 的二维数组，将其中一行的元素设置为另一行对应元素的两倍。

8.创建一个形状为 (4, 4) 的随机浮点数矩阵，将矩阵中所有小于 0.5 的元素设置为 0，大于等于 0.5 的元素设置为 1。