## Python Numpy

# Array index

```
import numpy as np

# Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

# Two ways of accessing the data in the middle row of the array.
# Mixing integer indexing with slices yields an array of lower rank,
# while using only slices yields an array of the same rank as the
# original array:
row_r1 = a[1, :]    # Rank 1 view of the second row of a
row_r2 = a[1:2, :]  # Rank 2 view of the second row of a
print(row_r1, row_r1.shape)  # Prints "[5 6 7 8] (4,)"
print(row_r2, row_r2.shape)  # Prints "[[5 6 7 8]] (1, 4)"

# We can make the same distinction when accessing columns of an array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)  # Prints "[ 2  6 10] (3,)"
print(col_r2, col_r2.shape)  # Prints "[[ 2]
                             #          [ 6]
                             #          [10]] (3, 1)"
```

a = [1,2,3,4]的shape是（4，）

---

# Boolean Array indexing

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)   # Find the elements of a that are bigger than 2;
```

```
                    # this returns a numpy array of Booleans of the same
                    # shape as a, where each slot of bool_idx tells
                    # whether that element of a is > 2.

print(bool_idx)      # Prints "[[False False]
                    #          [ True  True]
                    #          [ True  True]]"

# We use boolean array indexing to construct a rank 1 array
# consisting of the elements of a corresponding to the True values
# of bool_idx
print(a[bool_idx])  # Prints "[3 4 5 6]"

# We can do all of the above in a single concise statement:
print(a[a > 2])     # Prints "[3 4 5 6]"
```

---

## Array math —— elementwise

```
import numpy as np


x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)


# Elementwise sum; both produce the array
# [[ 6.0  8.0]
#  [10.0 12.0]]
print(x + y)
print(np.add(x, y))


# Elementwise difference; both produce the array
# [[-4.0 -4.0]
#  [-4.0 -4.0]]
print(x - y)
print(np.subtract(x, y))


# Elementwise product; both produce the array
# [[ 5.0 12.0]
#  [21.0 32.0]]
```

```
print(x * y)
print(np.multiply(x, y))


# Elementwise division; both produce the array
# [[ 0.2         0.33333333]
#  [ 0.42857143  0.5       ]]
print(x / y)
print(np.divide(x, y))


# Elementwise square root; produces the array
# [[ 1.          1.41421356]
#  [ 1.73205081  2.        ]]
print(np.sqrt(x))
```

---

# Array math —— dot()

```
import numpy as np


x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])


v = np.array([9,10])
w = np.array([11, 12])


# Inner product of vectors; both produce 219
print(v.dot(w))
print(np.dot(v, w))


# Matrix / vector product; both produce the rank 1 array [29 67]
print(x.dot(v))
print(np.dot(x, v))


# Matrix / matrix product; both produce the rank 2 array
# [[19 22]
#  [43 50]]
print(x.dot(y))
print(np.dot(x, y))
```

---

## Array math —— sum()

```
import numpy as np

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

print(np.sum(x))  # Compute sum of all elements; prints "10"
print(np.sum(x, axis=0))  # Compute sum of each column; prints "[4 6]"
print(np.sum(x, axis=1))  # Compute sum of each row; prints "[3 7]"
```

---

## Array math —— axis

如果axis=0，则沿着纵轴进行操作；axis=1，则沿着横轴进行操作。

如果是多维的呢？可以总结为：设axis=i，则numpy沿着第i个下标变化的方向操作。

例如刚刚的例子，可以将表示为：data =[[a00, a01],[a10,a11]]，所以axis=0时，沿着第0个下标变化的方向操作，也就是a00->a10, a01->a11。

一个例子：

```
>>> data = np.random.randint(0, 5, [4,3,2,3])
>>> data
array([[[[4, 1, 0],
         [4, 3, 0]],
        [[1, 2, 4],
         [2, 2, 3]],
        [[4, 3, 3],
         [4, 2, 3]]],

       [[[4, 0, 1],
         [1, 1, 1]],
        [[0, 1, 0],
         [0, 4, 1]],
        [[1, 3, 0],
```

```
         [0, 3, 0]]],

       [[[3, 3, 4],
         [0, 1, 0]],
        [[1, 2, 3],
         [4, 0, 4]],
        [[1, 4, 1],
         [1, 3, 2]]],

       [[[0, 1, 1],
         [2, 4, 3]],
        [[4, 1, 4],
         [1, 4, 1]],
        [[0, 1, 0],
         [2, 4, 3]]]])
```

当axis=0时，numpy验证第0维的方向来求和，也就是第一个元素值=a0000+a1000+a2000+a3000=11,第二个元素=a0001+a1001+a2001+a3001=5，同理可得最后的结果如下：

```
>>> data.sum(axis=0)
array([[[11,  5,  6],
        [ 7,  9,  4]],

       [[ 6,  6, 11],
        [ 7, 10,  9]],

       [[ 6, 11,  4],
        [ 7, 12,  8]]])
```

原来shape是[4,3,2,3]，axis=0求和后，第一维没有了，shape变成[3, 2, 3]

---

# Broadcasting

Broadcasting is a powerful mechanism that allows numpy to work with arrays of different shapes when performing arithmetic operations. Frequently we have a smaller array and a larger array, and we want to use the smaller array multiple times to perform some operation on the larger array.

For example, suppose that we want to add a constant vector to each row of a matrix.

Naive implementation (loop is costly):

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   # Create an empty matrix with the same
shape as x

# Add the vector v to each row of the matrix x with an explicit
loop
for i in range(4):
    y[i, :] = x[i, :] + v

# Now y is the following
# [[ 2  2  4]
#  [ 5  5  7]
#  [ 8  8 10]
#  [11 11 13]]
print(y)
```

This works; however when the matrix `x` is very large, **computing an explicit loop in Python could be slow.** Note that adding the vector `v` to each row of the matrix `x` is equivalent to forming a matrix `vv` by stacking multiple copies of `v` vertically, then performing elementwise summation of `x` and `vv`.

Still naive implementation:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
vv = np.tile(v, (4, 1))   # Stack 4 copies of v on top of each
other
```

```
print(vv)                 # Prints "[[1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]
                          #          [1 0 1]]"
y = x + vv  # Add x and vv elementwise
print(y)  # Prints "[[ 2  2  4
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

Numpy broadcasting allows us to perform this computation without actually creating multiple copies of `v`.

Real broadcasting:

```
import numpy as np

# We will add the vector v to each row of the matrix x,
# storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  # Add v to each row of x using broadcasting
print(y)  # Prints "[[ 2  2  4]
          #          [ 5  5  7]
          #          [ 8  8 10]
          #          [11 11 13]]"
```

The line `y = x + v` works even though `x` has shape `(4, 3)` and `v` has shape `(3,)` due to broadcasting; this line works as if `v` actually had shape `(4, 3)`, where each row was a copy of `v`, and the sum was performed elementwise.

---

# Image operations

```
print(a.shape,a.dtype)
>>>(1080, 1440, 3),uint8
图像为1080*1440*3字节，高为1080像素，宽为1440像素，3为通道数。
R,G,B用二进制的8位来表示(uint8)。
```

通道数为1是黑白图像。