创建 numpy.array

In [1]:

```
import numpy as np
```

使用 np.array 创建

在np.array()函数中传入一个python的list，python的list可以包含不同的元素，如一个python的list可以既包含数字又包含字符串，但是numpy的array只能包含一种类型的元素。 jupyter notebook中使用shift+tab可以查看函数的描述

In [11]:

```
array = np.array([i for i in range(1, 10)])
array
```

Out[11]:

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

使用其他方法创建

np.zeros()

In [12]:

```
array = np.zeros(10, dtype=float)
array
```

Out[12]:

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

In [15]:

```
array = np.zeros(shape=(3, 5), dtype=int)
array
```

Out[15]:

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

np.ones() 使用方法同上

np.full

In [17]:

```
array = np.full(shape=(3,5),fill_value=6,dtype=int)
array
```

Out[17]:

```
array([[6, 6, 6, 6, 6],
       [6, 6, 6, 6, 6],
       [6, 6, 6, 6, 6]])
```

`np.arange()`

In [18]:

```
array = np.arange(start=0,stop=10,step=2,dtype=int)
array
```

Out[18]:

```
array([0, 2, 4, 6, 8])
```

`np.arange(0,20,2)` 同python中 `[i for i in range(0,20,2)]` 作用相同。与python不同的情况是python中的range方法的步长不可以为浮点数，但是numpy的arange方法步长可以为浮点数

In [19]:

```
[i for i in range(0,2,0.2)]
```

```
--------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
<ipython-input-19-870e2053cf66> in <module>
----> 1 [i for i in range(0,2,0.2)]

TypeError: 'float' object cannot be interpreted as an integer
```

In [21]:

```
array = np.arange(0,2,0.2)
array
```

Out[21]:

```
array([0. , 0.2, 0.4, 0.6, 0.8, 1. , 1.2, 1.4, 1.6, 1.8])
```

`np.linspace`

linspace其实是lineaspace的缩写。大体用法与arange一致。np.linspace(0,20,10)的作用是在 `[0,20]` 取出距离相同的10个点（包含0和20）。

In [22]:

```
array = np.linspace(start=0,stop=20,num=11)
array
```

Out[22]:

```
array([ 0.,  2.,  4.,  6.,  8., 10., 12., 14., 16., 18., 20.])
```

random

In [23]:

```
np.random.randint(low=0,high=10,size=(3,5)) # randint 使用
```

Out[23]:

```
array([[9, 6, 9, 0, 7],
       [8, 2, 9, 2, 8],
       [7, 9, 0, 2, 1]])
```

In [30]:

```
np.random.seed(222) # 设置随机数种子
np.random.randint(10)
```

Out[30]:

```
6
```

In [31]:

```
np.random.seed(222) # 设置随机数种子
np.random.randint(10) # 两次结果相同
```

Out[31]:

```
6
```

random的使用方法同 `randint` 的使用方法相同，只是random返回的是浮点数

`random.normal()` 使用

In [33]:

```
np.random.normal(10,100,size=(3,5)) # 生成size形状的符合正态分布的随机数
```

Out[33]:

```
array([[  30.94302342, -135.87337403,   38.85385311,   58.20064448,
         111.46229453],
       [  69.66489147,  -65.17306452,  -15.82410734,   46.0827302 ,
        -127.53216492],
       [ -98.62401756,  -16.48654603,   -3.31909479,  143.91426009,
         -17.1816209 ]])
```

In [34]:

```
np.random.normal(0,1,10)
```

Out[34]:

```
array([ 1.12479375,  2.16500669,  1.37172051,  0.04297725, -1.319879
85,
       -0.49967542, -2.17138661, -1.37122129, -0.17276819,  0.234029
])
```

Jupyter notebook 帮助命令

In [35]:

```
np.array?
```

In [36]:

```
help(np.arange)
```

```
Help on built-in function arange in module numpy:

arange(...)
    arange([start,] stop[, step,], dtype=None)

    Return evenly spaced values within a given interval.

    Values are generated within the half-open interval ``[start, sto
p)``
    (in other words, the interval including `start` but excluding `s
top`).
    For integer arguments the function is equivalent to the Python b
uilt-in
    `range` function, but returns an ndarray rather than a list.

    When using a non-integer step, such as 0.1, the results will oft
en not
    be consistent.  It is better to use `numpy.linspace` for these c
ases.

    Parameters
    ----------
    start : number, optional
        Start of interval.  The interval includes this value.  The d
efault
        start value is 0.
    stop : number
        End of interval.  The interval does not include this value,
except
        in some cases where `step` is not an integer and floating po
int
        round-off affects the length of `out`.
    step : number, optional
        Spacing between values.  For any output `out`, this is the d
istance
        between two adjacent values, ``out[i+1] - out[i]``.  The def
ault
        step size is 1.  If `step` is specified as a position argume
nt,
        `start` must also be given.
    dtype : dtype
        The type of the output array.  If `dtype` is not given, infe
r the data
        type from the other input arguments.

    Returns
    -------
    arange : ndarray
        Array of evenly spaced values.

        For floating point arguments, the length of the result is
        ``ceil((stop - start)/step)``.  Because of floating point ov
erflow,
        this rule may result in the last element of `out` being grea
ter
        than `stop`.

    See Also
    --------
    linspace : Evenly spaced numbers with careful handling of endpoi
nts.
```

```
    ogrid: Arrays of evenly spaced numbers in N-dimensions.
    mgrid: Grid-shaped arrays of evenly spaced numbers in N-dimensio
ns.

    Examples
    --------
    >>> np.arange(3)
    array([0, 1, 2])
    >>> np.arange(3.0)
    array([ 0.,  1.,  2.])
    >>> np.arange(3,7)
    array([3, 4, 5, 6])
    >>> np.arange(3,7,2)
    array([3, 5])
```

numpy.array() 基本操作

Reshape

In [8]:

```
x = np.arange(12)
x
```

Out[8]:

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

In [9]:

```
X = x.reshape((3,-1)) # 指定第一个维度，第二个维度自动计算
X
```

Out[9]:

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

基本属性

1. ndim # 返回的是一个数字，表示有多少个维度
2. shape # 返回的是一个tuple，tuple表示每个维度具体有多少个元素
3. size # 表示共有多少个元素

In [4]:

```
X.ndim
```

Out[4]:

```
2
```

In [5]:

```
X.shape
```

Out[5]:

```
(2, 5)
```

In [6]:

```
X.size
```

Out[6]:

```
10
```

数据访问

对于一维数组，访问方法与python访问方法相同，就常见的索引+切片操作

对于多维数组，使用形如 `X[2,2]` 或 `X[(2,2)]` 这种方式访问

In [10]:

```
X[2,2]
```

Out[10]:

```
10
```

多维数组的切片操作

多维数组的访问格式为 `X[start:stop:step,start:stop:step,.....]` 以此类推

In [23]:

```
X[:2, ::2] # 前两行，每隔两列
```

Out[23]:

```
array([[0, 2],
       [4, 6]])
```

改变子数组的元素会改变到原数组，若不想改变原数组，使用copy()操作

In [24]:

```
X
```

Out[24]:

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

In [25]:

```
sub_X = X[:2,:3]
sub_X
```

Out[25]:

```
array([[0, 1, 2],
       [4, 5, 6]])
```

In [26]:

```
sub_X[0,0] = 6
X # X的数值被改变
```

Out[26]:

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

In [27]:

```
sub_X = X[:2,:3].copy()
sub_X
```

Out[27]:

```
array([[6, 1, 2],
       [4, 5, 6]])
```

In [29]:

```
sub_X[0,0] = 100000
X # X的值不会发生改变
```

Out[29]:

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

合并操作

`concatenate`

concatenate的参数主要是待拼接元素的tuple和拼接的轴(axis)。可以简单将不指定axis的参数或者axis=0时理解为样本的拼接，当axis=1的时候理解为样本特征的拼接。

一维数组拼接

In [34]:

```
x = np.random.randint(0, 10, 3)
x
y = np.random.randint(0, 10, 3)
np.concatenate((x, y))
```

Out[34]:

```
array([1, 6, 6, 9, 1, 3])
```

In [36]:

```
x = np.random.randint(0,10,(3,4))
y = np.random.randint(0,10,(3,4))
np.concatenate((x,y)) # 看作是样本的拼接
```

Out[36]:

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

In [37]:

```
x = np.random.randint(0,10,(3,4))
y = np.random.randint(0,10,(3,4))
np.concatenate((x,y),axis=1) # 看作是特征的拼接
```

Out[37]:

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

concatenate只能拼接维度相同的元素

In [38]:

```
x = np.random.randint(0,10,(3,4))
y = np.random.randint(0,10,(2,6))
np.concatenate((x,y))
```

```
--------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-38-9cd098e21b9f> in <module>
      1 x = np.random.randint(0,10,(3,4))
      2 y = np.random.randint(0,10,(2,6))
----> 3 np.concatenate((x,y))

ValueError: all the input array dimensions except for the concatenat
ion axis must match exactly
```

`vstack`

v表示vertical，表示在垂直方向上进行堆叠。因为concatenate方法只能处理维度相同的元素。若有形如 `[[1,2,3],[4,5,6]]` 和 `[7,8,9]` 进行合并的时候必须要将 `[7,8,9]` 进行reshape才可以，但是vstack可以直接进行拼接。

In [39]:

```
x = np.array([[1,2,3],[4,5,6]])
y = np.array([7,8,9])
np.concatenate((x,y)) # 维度不同会报错
```

```
---------------------------------------------------------------------
-------
ValueError                                Traceback (most recent cal
l last)
<ipython-input-39-e9c5e97afc8e> in <module>
      1 x = np.array([[1,2,3],[4,5,6]])
      2 y = np.array([7,8,9])
----> 3 np.concatenate((x,y)) # 维度不同会报错

ValueError: all the input arrays must have same number of dimensions
```

In [41]:

```
y = y.reshape(-1,3)
np.concatenate((x,y)) # 经过reshape后改造成维度相同的元素则不会报错
```

Out[41]:

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

In [43]:

```
z = np.array([9,10,11])
np.vstack((x,z)) # 使用vstack不需要进行reshape操作，可以直接进行堆叠
```

Out[43]:

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

hstack

This is equivalent to concatenation along the second axis, except for 1-D arrays where it concatenates along the first axis. Rebuilds arrays divided by hsplit.
具体操作和stack相对应，vstack的的定义也和上面的定义相对应

分割操作

`split`

共有三个参数。第一个是表示被分割的元素，第二个参数是一个list表示分割点。如np.split(x, `[2,3]`)表示将x分割为三段.第三个参数是分割轴

In [44]:

```
x = np.arange(0,10)
x
```

Out[44]:

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

In [46]:

```
x1, x2, x3 = np.split(x,[2,6])
```

In [50]:

```
x1 # x1为[0,2)的元素
```

Out[50]:

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

In [51]:

```
x2 # x2为[2,6)的元素
```

Out[51]:

```
array([2, 3, 4, 5])
```

In [52]:

```
x3 # x3为[6,-1]的元素
```

Out[52]:

```
array([6, 7, 8, 9])
```

In [53]:

```
X = np.arange(16).reshape(4,4)
X
```

Out[53]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [54]:

```
X1,X2 = np.split(X,[2])
```

In [55]:

```
X1
```

Out[55]:

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [56]:

```
X2
```

Out[56]:

```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [57]:

```
X1,X2 = np.split(X,[2],axis=1)
```

In [58]:

```
X1
```

Out[58]:

```
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
```

In [59]:

```
X2
```

Out[59]:

```
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

`hsplit`

hsplit相当与指定了axis=1

In [60]:

```
X1, X2 = np.hsplit(X, [2])
```

In [61]:

```
X1
```

Out[61]:

```
array([[ 0,  1],
       [ 4,  5],
       [ 8,  9],
       [12, 13]])
```

In [62]:

```
X2
```

Out[62]:

```
array([[ 2,  3],
       [ 6,  7],
       [10, 11],
       [14, 15]])
```

`vsplit`

vsplit相当与指定了axis=0

In [63]:

```
X1, X2 = np.vsplit(X, [2])
```

In [64]:

```
X1
```

Out[64]:

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [65]:

```
X2
```

Out[65]:

```
array([[ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

运算操作

- **python的list因为可以包含不同的元素，因此对python的list与n相乘，得到的结果是将list的内容重复n次**
- **numpy的array若只包含数字，对numpy的array与n相乘，得到的结果是array的每一个数字都乘以n**

In [66]:

```
x = [i for i in range(10)]
x*2
```

Out[66]:

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

In [67]:

```
y = np.arange(10)
y*2
```

Out[67]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

性能比较

In [70]:

```
n = 1000000
L = [i for i in range(n)]
```

In [71]:

```
%%time
A = [2*e for e in L]
```

```
CPU times: user 54.6 ms, sys: 18 ms, total: 72.5 ms
Wall time: 72.2 ms
```

In [72]:

```
L = np.arange(n)
```

In [73]:

```
%%time
A = np.array(2*e for e in L)
```

```
CPU times: user 208 µs, sys: 182 µs, total: 390 µs
Wall time: 5.64 ms
```

universal function

In [74]:

```
X = np.arange(1, 16).reshape((3, 5))
X
```

Out[74]:

```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10],
       [11, 12, 13, 14, 15]])
```

In [75]:

```
X+1
```

Out[75]:

```
array([[ 2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11],
       [12, 13, 14, 15, 16]])
```

其余四则操作同理

In [76]:

```
np.sin(X)
```

Out[76]:

```
array([[ 0.84147098,  0.90929743,  0.14112001, -0.7568025 , -0.95892
427],
       [-0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849, -0.54402
111],
       [-0.99999021, -0.53657292,  0.42016704,  0.99060736,  0.65028
784]])
```

In [77]:

```
np.power(2,X)
```

Out[77]:

```
array([[    2,     4,     8,    16,    32],
       [   64,   128,   256,   512,  1024],
       [ 2048,  4096,  8192, 16384, 32768]])
```

In [78]:

```
np.exp(X)
```

Out[78]:

```
array([[2.71828183e+00, 7.38905610e+00, 2.00855369e+01, 5.45981500e+
01,
        1.48413159e+02],
       [4.03428793e+02, 1.09663316e+03, 2.98095799e+03, 8.10308393e+
03,
        2.20264658e+04],
       [5.98741417e+04, 1.62754791e+05, 4.42413392e+05, 1.20260428e+
06,
        3.26901737e+06]])
```

In [79]:

```
np.log(X)
```

Out[79]:

```
array([[0.        , 0.69314718, 1.09861229, 1.38629436, 1.60943791],
       [1.79175947, 1.94591015, 2.07944154, 2.19722458, 2.30258509],
       [2.39789527, 2.48490665, 2.56494936, 2.63905733, 2.7080502 
]])
```

矩阵运算

In [80]:

```
A = np.arange(4).reshape(2, 2)
A
```

Out[80]:

```
array([[0, 1],
       [2, 3]])
```

In [81]:

```
B = np.full((2, 2), 10)
B
```

Out[81]:

```
array([[10, 10],
       [10, 10]])
```

普通四则符号是两个矩阵对应的元素进行四则操作

In [82]:

```
B-A
```

Out[82]:

```
array([[10,  9],
       [ 8,  7]])
```

其余四则同理

In [83]:

```
A.dot(B) # 内积
```

Out[83]:

```
array([[10, 10],
       [50, 50]])
```

聚合操作(求和、求均值等)

sum()

In [84]:

```
big_array = np.random.rand(1000000)
%timeit sum(big_array) # python内置函数
%timeit np.sum(big_array) # numpy函数
```

```
145 ms ± 3.82 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
353 µs ± 17 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

`min()` `max()`

In [85]:

```
np.min(big_array)
```

Out[85]:

```
1.5158657894476235e-06
```

In [86]:

```
np.max(big_array)
```

Out[86]:

```
0.9999996888614331
```

也可以直接使用numpy中array对象中的min和max函数

In [88]:

```
big_array.min()
```

Out[88]:

```
1.5158657894476235e-06
```

多维数组的聚合操作

In [89]:

```
X = np.arange(16).reshape(4,-1)
X
```

Out[89]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [90]:

```
np.sum(X)
```

Out[90]:

```
120
```

In [91]:

```
np.sum(X, axis=0)
```

Out[91]:

```
array([24, 28, 32, 36])
```

In [92]:

```
np.sum(X, axis=1)
```

Out[92]:

```
array([ 6, 22, 38, 54])
```

NUMPY AXES EXPLAINED (https://www.sharpsightlabs.com/blog/numpy-axes-explained/)

In [96]:

```
np.mean(X)
```

Out[96]:

```
7.5
```

In [97]:

```
np.mean(X,axis=0) # 按列求均值
```

Out[97]:

```
array([6., 7., 8., 9.])
```

In [98]:

```
np.mean(X, axis=1) # 按行求均值
```

Out[98]:

```
array([ 1.5,  5.5,  9.5, 13.5])
```

In [99]:

```
np.median(X) # 中位数
```

Out[99]:

```
7.5
```

In [100]:

```
x = np.random.normal(0,1,10)
```

In [101]:

```
np.std(x)
```

Out[101]:

```
0.5546352436742645
```

In [102]:

```
np.var(x)
```

Out[102]:

```
0.30762025352561084
```

索引

In [103]:

```
x = np.random.normal(0, 1, 1000000)
```

In [104]:

```
np.argmax(x)
```

Out[104]:

```
58708
```

In [105]:

```
x[58708]
```

Out[105]:

```
4.693375882092018
```

In [106]:

```
np.max(x)
```

Out[106]:

```
4.693375882092018
```

In [107]:

```
x = np.arange(16)
```

In [111]:

```
x.sort() # 将x排序
x
```

Out[111]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 1
5])
```

In [122]:

```
np.random.shuffle(x) # 将x打乱
x
```

Out[122]:

```
array([ 1,  0, 14,  5,  9, 12,  3, 11, 10,  2, 15,  8,  6,  7, 13,
4])
```

In [114]:

```
x.argsort() # x排序的下标
```

Out[114]:

```
array([ 5, 15,  3,  6,  0,  2,  9, 11,  1,  7,  4, 12,  8, 14, 13, 1
0])
```

In [128]:

```
x
```

Out[128]:

```
array([ 1,  0, 14,  5,  9, 12,  3, 11, 10,  2, 15,  8,  6,  7, 13,
4])
```

In [134]:

```
np.partition(x, 8)  # 以排序后的第8个数进行区分
```

Out[134]:

```
array([ 3,  1,  2,  0,  4,  7,  6,  5,  8, 11,  9, 10, 12, 14, 13, 1
5])
```

In [117]:

```
np.random.shuffle(x) # 将x的顺序打乱
X = x.reshape(4,4)
X
```

Out[117]:

```
array([[ 1, 10, 14,  8],
       [ 3,  2,  4, 15],
       [13,  7,  9,  5],
       [12, 11,  6,  0]])
```

In [118]:

```
np.sort(X,axis=1) # 列排序。行排序同理
X
```

Out[118]:

```
array([[ 1, 10, 14,  8],
       [ 3,  2,  4, 15],
       [13,  7,  9,  5],
       [12, 11,  6,  0]])
```

Fancy Indexing

Fancy Indexing在一维数组中

In [142]:

```
x = np.arange(16)
ind = [2,4,5]
x
```

Out[142]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 1
5])
```

In [143]:

```
x[ind]
```

Out[143]:

```
array([2, 4, 5])
```

Fancy在多维数组

In [144]:

```
X = x.reshape(4, -1)
X
```

Out[144]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

In [145]:

```
row = np.array([0, 1, 2])
col = np.array([1, 2, 3])
X[row, col]
```

Out[145]:

```
array([ 1,  6, 11])
```

根据布尔值取值

In [146]:

```
x
```

Out[146]:

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 1
5])
```

In [147]:

```
x < 6
```

Out[147]:

```
array([ True,  True,  True,  True,  True,  True, False, False, Fals
e,
       False, False, False, False, False, False, False])
```

其他比较符号如（!=，==，<=）等符号也同理

In [148]:

```
2 * x == 24 - 4 * x
```

Out[148]:

```
array([False, False, False, False,  True, False, False, False, Fals
e,
       False, False, False, False, False, False, False])
```

In [149]:

```
np.sum(x <= 3)
```

Out[149]:

```
4
```

In [151]:

```
x[x < 5]
```

Out[151]:

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

In [152]:

```
x[x % 2 == 0]
```

Out[152]:

```
array([ 0,  2,  4,  6,  8, 10, 12, 14])
```

In [153]:

```
X[X[:,3] % 3 == 0, :]
```

Out[153]:

```
array([[ 0,  1,  2,  3],
       [12, 13, 14, 15]])
```

In [ ]:

```

```