NumPy

Array Commands

- **Simple array creation**
  - **a = array([0,1,2,3])**
  - **a**
  - array([0, 1, 2, 3])
- **Checking the Type**
  - **type(a)**
  - <type 'array'>
- **Numeric Type of elements**
  - **a.dtype**
  - dtype(‘int32’)
- **Bytes per element**
  - **a.itemsize**
  - dtype(‘int32’)
  - 4

Array Commands

- Array Shape

#returns a tuple listing the length of the array along each dimension.

  - a.shape
  - Shape(a)
  - (4,)

- Array Size

#reports the entire number of elements in an array.

  - a.size
  - 4
  - size(a)
  - 4

Setting Array Elements

- Array Indexing
  - a = array([0,1,2,3])
  - a[0]
  - Returns 0
  - a[0] = 10
  - a
  - [10, 1, 2, 3]
- Fill   #set all values in an array.
  - a.fill(0)
  - a
  - [0, 0, 0, 0]
  - a[:] = 1   # same but slower
  - a
  - [1,1,1,1]

Setting Array Elements

- # assigning a float to into an int32 array will truncate decimal part.
  - a[0] = 10.6
  - a
  - [10, 1, 2, 3]

Multi-Dimensional Arrays

- a = array([[0, 1, 2, 3],
- [10,11,12,13]])
- a
- array([[0, 1, 2, 3],
- [10,11,12,13]])
- (Rows,Columns)
- a = array([[0, 1, 2, 3],
- [10,11,12,13]])
  - a.shape
  - (2, 4)
- Element Count
  - a.size
    - Shows 8
- a.ndims
    - Shows 2

Setting Array Elements

- Get/Set Elements
- a = array([[0, 1, 2, 3], [10,11,12,13]])
    - a[1,3]
    - Shows 13
    - a[1,3] = -1
    - a
    - array([[0, 1, 2, 3],
  - [10,11,12,-1]])
- ADDRESS FIRST ROW USING SINGLE INDEX
  - a[1]
  - Shows
  - array([10, 11, 12, -1])

Array Slicing

- #Works like standard python slicing
- a[0,3:5]
- array([3, 4])
- a[4:,4:]
- array([[44, 45],
-        [54, 55]])
- a[:,2]
- array([2,12,22,32,42,52])
- #strides
- a[2::2,::2]
- array([[20, 22, 24],
-        [40, 42, 44]])

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

Array Striding

numpy.ndarray.strides

- Tuple of bytes to step in each dimension when traversing an array.
- The strides of an array tell us how many bytes we have to skip in memory to move to the next position along a certain axis.
- For example, we have to skip 4 bytes (1 value) to move to the next column, but 24 (6*4) bytes (6 values) to get to the same position in the next row.
- As such, the strides for the array arr with shape 6,6 will be 24,4 as 20 bytes(6 values) to get to the same position in the next row and 4 .bytes (1 value) to move to the next column,

Memory Model

Print(a.strides)

(24, 8)

print a.flags.fortran, a.flags.contiguous

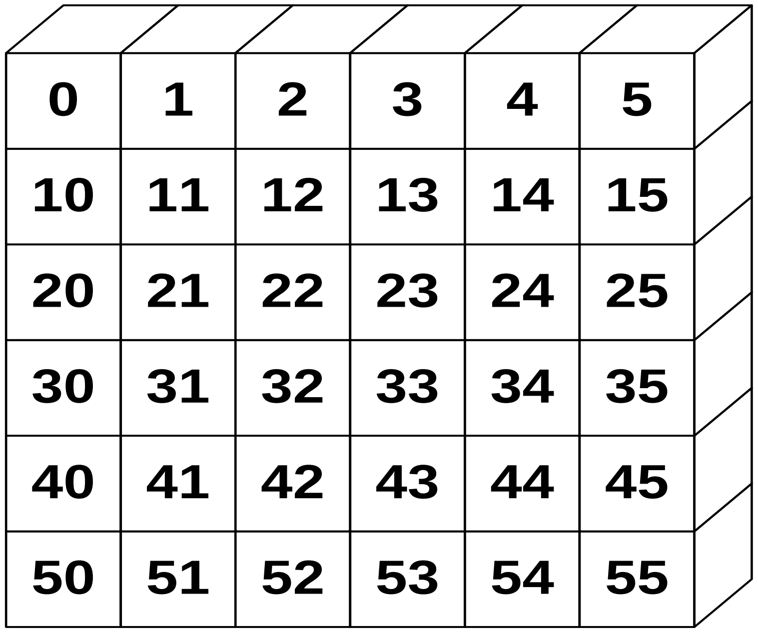
False True

print a.T.strides

(8, 24)

>>> print a.T.flags.fortran, a.T.flags.contiguous

True False



- Every dimension of an ndarray is accessed by stepping (striding) a fixed number of bytes through memory.
- If memory is contiguous, then the strides are “pre-computed” indexing-formulas for either Fortran-order (first-dimension varies the fastest), or C-order (last-dimension varies the fastest) arrays.

Array Slicing (Views)

- Memory model allows “simple indexing” (integers and slices) into the array to be a **view** of the same data. Slices are references to memory in original array. Changing values in a slice also changes the original array.
- a = np.array([[1, 2, 3],
- [4,5,6]])

```
b = a[:,::2]
# b becomes [[1,3],[4,6]]
b[0,1] = 100
Print (a)    # a changes
[[   1    2  100]]
 [   4    5    6]]
c = a[:,::2].copy()
# c becomes [[1,00],[4,6]]
c[1,0] = 500
print a
[[   1.    2.  100.]]
 [   4.    5.    6.]]
```

Slice is a view

- a = array([0,1,2,3,4])

- # create a slice
- b = a[2:4]
- Print(b) shows
- [2,3])
- b[0] = 10
- B shows [10,3]
- # changing b changed a!
- Print(a)
- array([1, 2, 10, 3, 4])

Fancy Indexing

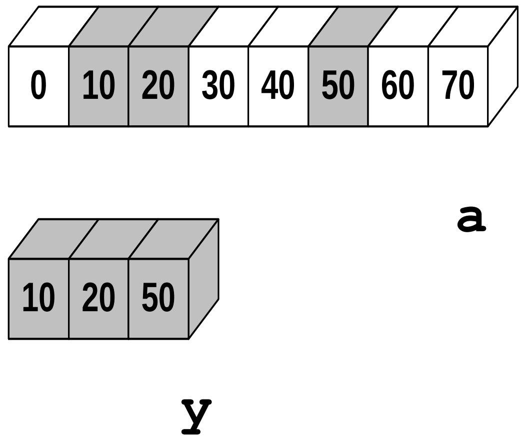
- Fancy indexing is like the simple indexing we've already seen, but we pass arrays of indices in place of single scalars. This allows us to very quickly access and modify complicated subsets of an array's values.
- Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once.
- Suppose we want to access three different elements. We could do it like this:
- If x=[0,10,20,30,40,50,60,70,80]
- [x[1],x[5],x[6]] would show 10 and 50 and 60
- Alternatively, we can pass a single list or array of indices to obtain the same result: Ind=x[1,5,6]

Fancy Indexing

- Indexing by Position
- a = arange(0,80,10)

[0 10 20 30 40 50 60 70]

- y = a[[1, 2, -3]]
- print y
- [10 20 50]
- # using take
- y = take(a,[1,2,-3])
- print y
- [10 20 50]



Indexing with position

- mask = np.array([0,1,1,0,0,1,0,0], dtype=bool)
- Array a is [0 10 20 30 40 50 60 70]

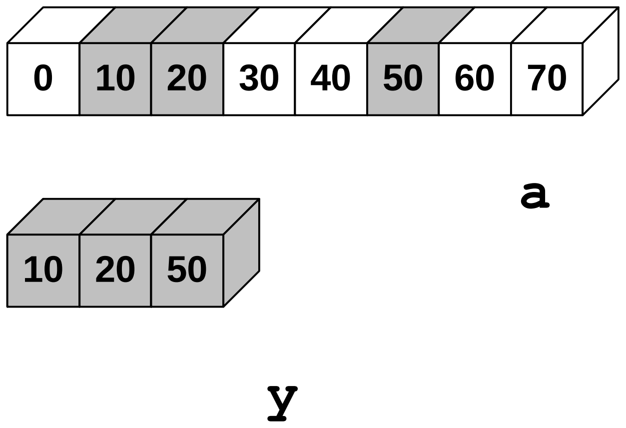
fancy indexing

- y = a[mask]
- Print(y)
- [10,20,50]

using compress

y = compress(mask, a)

- print y
- [10,20,50]



Fancy Indexing

Fancy indexing also works in multiple dimensions. Consider the following array:

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

Like with standard indexing, the first index refers to the row, and the second to the column:

- Row=np.array([0,1,2])
- Col=np.array([2,1,3])
- X[row,col]   # shows ([2,5,11])
- The first value in the result is x[0,2], second is x[1,1] and 3rd is x[2,3]
- One to One is done in pairing of indexes following broadcasting rule.

Combining Fancy Indexing with other schemes

- We can combine fancy and simple indices:
- [[0,1,2,3],
- [4,5,6,7],
- [8,9,10,11]]
- X[2,[2,0,1]]   # means row 2 and 2^{nd} , 0^{th} and 1^{st} element of the row
- Shows
- [10,8,9]

Combining Fancy Indexing with other schemes

- We can also combine fancy indexing with slicing
- [[0,1,2,3],[4,5,6,7],[8,9,10,11]]
- x[1:,[2,0,1])
- Shows
- Array[[6,4,5],[10,8,9]]

Modifying Values with Fancy Indexing

- X=np.array([0,1,2,3,4,5,6,7,8,9])
- i = np.array([2, 1, 8, 4])
- x[i] = 99
- print(x)
- [0 99 99 3 99 5 6 7 99 9]

Fancy Indexing in 2-D

- Unlike slicing, fancy indexing creates copies instead of views into original arrays.

0	1	2	3	4	5
10	11	12	13	14	15
20	21	22	23	24	25
30	31	32	33	34	35
40	41	42	43	44	45
50	51	52	53	54	55

```
arr=np.array([[0,1,2,3,4,5],
 [10,11,12,13,14,15],
  [20,21,22,23,24,25],
  [30,31,32,33,34,35],
  [40,41,42,43,44,45],
  [50,51,52,53,54,55]])
#accessing one to one mapping, 0,1    1,2   2,3   3,4   4,5
arr[(0,1,2,3,4),(1,2,3,4,5)]
```

shows

- array([1, 12, 23, 34, 45])

Fancy Indexing in 2-D

Example :

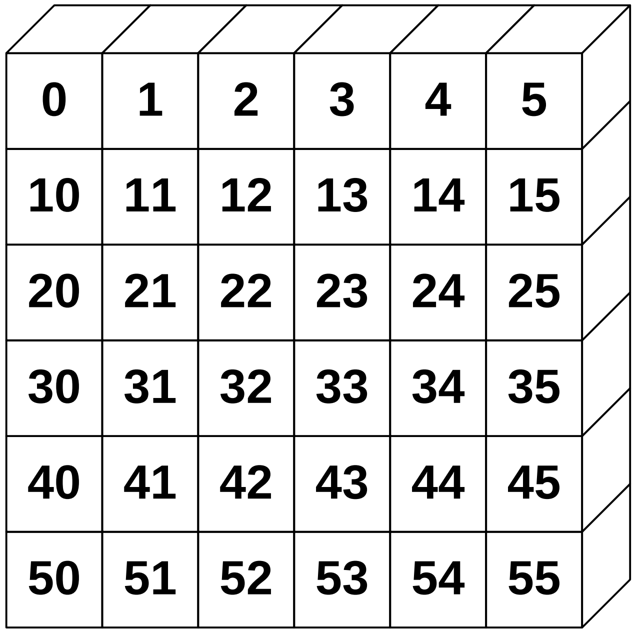
a[3:,[0, 2, 5]]

#column 0 2 and 5 are required staring row 3

array([[30, 32, 35],

      [40, 42, 45]])

      [50, 52, 55]])



Array Calculation Methods

- a = array([[1,2,3], [4,5,6]], float)

Sum defaults to summing all array values.

- sum(a)

21.

supply the keyword axis to sum along the 0th axis.

- sum(a, axis=0)

array([5., 7., 9.])

supply the keyword axis to sum along the last axis.

- sum(a, axis=-1)

array([6., 15.])

Sum Array Method

- # The a.sum() defaults to summing *all* array values
- a = array([[1,2,3], [4,5,6]], float)
- a.sum()

21.

- # Supply an axis argument to sum along a specific axis.
- a.sum(axis=0)

array([5., 7., 9.])

Product

- # product along columns.
- a = array([[1,2,3], [4,5,6]], float)
- a.prod(axis=0)
- array([4., 10., 18.])

Min/Max

- a = array([2.,3.,0.,1.])
- a.min(axis=0)

0.

- # use Numpy’s amin() instead of Python’s builtin min()
- # for speed operations on multi-dimensional arrays.
- amin(a, axis=0)

0.

Flags

- import numpy as np
- x=np.array([[3,4],[3,5]])
- print(x.flags)

SHOWS

C_CONTIGUOUS : True

F_CONTIGUOUS : False

OWNDATA : True

WRITEABLE : True

ALIGNED : True

WRITEBACKIFCOPY : False

UPDATEIFCOPY : False

Flags

C_CONTIGUOUS (C)	The data is in a single, C-style contiguous segment.
F_CONTIGUOUS (F)	The data is in a single, Fortran-style contiguous segment.
OWNDATA (O)	The array owns the memory it uses or borrows it from another object.
WRITEABLE (W)	The data area can be written to. Setting this to False locks the data, making it read-only. A view (slice, etc.) inherits WRITEABLE from its base array at creation time, but a view of a writeable array may be subsequently locked while the base array remains writeable. (The opposite is not true, in that a view of a locked array may not be made writeable. However, currently, locking a base object does not lock any views that already reference it, so under that circumstance it is possible to alter the contents of a locked array via a previously created writeable view onto it.) Attempting to change a non-writeable array raises a RuntimeError exception.
ALIGNED (A)	The data and all elements are aligned appropriately for the hardware.

Flags

WRITEBACKIFCOPY (X)	This array is a copy of some other array. The C-API function PyArray_ResolveWritebackIfCopy must be called before deallocating to the base array will be updated with the contents of this array.
UPDATEIFCOPY (U)	(Deprecated, use WRITEBACKIFCOPY) This array is a copy of some other array. When this array is deallocated, the base array will be updated with the contents of this array.
FNC	F_CONTIGUOUS and not C_CONTIGUOUS.
FORC	F_CONTIGUOUS or C_CONTIGUOUS (one-segment test).
BEHAVED (B)	ALIGNED and WRITEABLE.
CARRAY (CA)	BEHAVED and C_CONTIGUOUS.
FARRAY (FA)	BEHAVED and F_CONTIGUOUS and not C_CONTIGUOUS.

Clip

- Limit values to a range

- >>> a = array([[1,2,3],
-                 [4,5,6]], float)

- # Set values < 3 equal to 3.
- # Set values > 5 equal to 5.
- >>> a.clip(3,5)
- >>> a
- array([[3., 3., 3.],
-        [4., 5., 5.]])

Clip

- Clip (limit) the values in an array.
- Given an interval, values outside the interval are clipped to the interval edges.
- For example, if an interval of [0,1]
- Given an interval, values outside the interval are clipped to the interval edges.
- a=np.arange(10) generates a[0,1,2,3,4,5,6,7,8,9]
- Print(np.clip(a,1,8)
- print(a)
- [1,1,2,3,4,5,6,7,8,8]
- Np.clip(a,3,6,out=a)
- [3 3 3 3 4 5 6 6 6 6]

ptp

- Peak-to-peak (maximum - minimum) value along the given axis.
- # Calculate max – min for array along columns
- a = array([[1,2,3],
- [4,5,6]], float)
- a.ptp(axis=0)
- array([3.0, 3.0, 3.0])
- max – min for entire array.
- a.ptp(axis=None)
- 5.0

ptp

- x=np.array(np.arange(12).reshape((3,4)))
- Print(x) shows [[0 1 2   3]
  [4 5 6   7]
  [8 9 10 11]]
- X.ptp(0)   # (8-0, 9-1, 10-2, 11-3)
- Shows [8 8 8 8]
- Print(x.ptp(1)) #(3-0, 7-4, 11-8)
- Shows [3 3 3]
- Print(x.ppt)
- Shows 11   # (11-0)

Random in numpy

- You can also create an array where each element is a random number using numpy.random.rand.
- np.random.rand(3,4)
- array([[0.2247223 , 0.92240549, 0.14541893, 0.61731257],
- [0.00154957, 0.82342197, 0.74044906, 0.11466845],
- [0.6152478 , 0.14433138, 0.13009583, 0.22981301]])
- Creating arrays full of random numbers can be useful when you want to quickly test your code with sample arrays.

N-Dimensional NumPy Arrays

- This doesn't happen extremely often, but there are cases when you'll want to deal with arrays that have greater than 3 dimensions. One way to think of this is as a list of lists of lists.
- Let's say we want to store the monthly earnings of a store, but we want to be able to quickly lookup the results for a quarter, and for a year. The earnings for one year might look like this:
- [500,505,490,810,450,678,234,897,430,560,1023,640)
- The store earned $500 in January, $505 February, and so on. We can split up these earnings by quarter into a list of lists:
- year_one = [[500,505,490], [810,450,678], [234,897,430], [560,1023,640]]

N-Dimensional NumPy Arrays

- We can retrieve the earnings from January by calling year_one[0][0].
- If we want the results for a whole quarter, we can call year_one[0] or year_one[1].
- We now have a 2-dimensional array, or matrix. But what if we now want to add the results from another year? We have to add a third dimension:
- earnings = [

```
[ [500,505,490], [810,450,678], [234,897,430], [560,1023,640] ],
[ [600,605,490], [345,900,1000], [780,730,710], [670,540,324] ]
]
```

N-Dimensional NumPy Arrays

- We can retrieve the earnings from January of the first year by calling
  earnings[0][0][0] .
- We now need three indexes to retrieve a single element.
- A three-dimensional array in NumPy is much the same. In fact, we can convert earnings to an array and then get the earnings for January of the first year:

```
earnings = np.array(earnings)
earnings[0,0,0]
```

It shows 500

- We can also find the shape of the array:

```
earnings.shape
```

it shows (2, 4, 3)

N-Dimensional NumPy Arrays

Indexing and slicing work the exact same way with a 3-dimensional array, but now we have an extra axis to pass in.

If we wanted to get the earnings for January of all years, we could do this:

earnings[:,0,0]

returns array([500, 600])

If we wanted to get first quarter earnings from both years, we could do this:

earnings[:,0,:]

array([[500, 505, 490], [600, 605, 490]])

N-Dimensional NumPy Arrays

- Adding more dimensions can make it much easier to query your data if it's organized in a certain way.
- As we go from 3-dimensional arrays to 4-dimensional and larger arrays, the same properties apply, and they can be indexed and sliced in the same ways.
- **Converting Data Types**
- You can use the numpy.ndarray.astype method to convert an array to a different type. The method will actually copy the array, and return a new array with the specified data type. For instance, we can convert wines to the int data type
- wines.astype(int)

- import numpy as np
- wines=np.genfromtxt("winequality-red.csv", delimiter=";",skip_header=1)
- wines

- array([[7.4 , 0.7 , 0. , ..., 0.56 , 9.4 , 5.], [7.8 , 0.88 , 0. , ..., 0.68 , 9.8 , 5.], [7 ..., 0.65 , 9.8 , 5.], ..., [6.3 , 0.51 , 0.13 , ..., 0.75 , 11. , 6.], [5.9 , 0.645, 0.12 ,], [6. , 0.31 , 0.47 , ..., 0.66 , 11. , 6.]])