Computer Vision

Numpy and basic linear algebra

Numpy

- Numpy array creation
- Array access and operations
- Basic linear algebra

Numpy

- Stands for Numerical Python
- Is the fundamental package required for high performance computing and data analysis
- It provides
  - ndarray for creating multiple dimensional arrays
  - Standard math functions for fast operations on entire arrays of data without having to write loops
  - Tools for reading/writing array data
  - Linear algebra tools
  - etc.

ndarray vs list of lists

- Say you have grades of three exams (2 midterms and 1 final) in a class of 10 students.
  - grades = [[79, 95, 60],
    [95, 60, 61],
    [99, 67, 84],
    [76, 76, 97],
    [91, 84, 98],
    [70, 69, 96],
    [88, 65, 76],
    [67, 73, 80],
    [82, 89, 61],
    [94, 67, 88]]

- How to get final exam grade of student 0?
  - grades[0][2]
- How to get grades of student 2?
  - grades[2]
- How to get grades of all students in midterm 1?
- How to get midterm grades of the first three students (or all female students, or those who failed final)?
- How to get mean grade of each exam?
- How to get (weighted) average exam grade for each student?

ndarray vs list of lists

- gArray = array(examGrades)

```
In [3]: gArray
Out[3]:
array([[79, 95, 60],
[95, 60, 61],
[99, 67, 84],
...,
[67, 73, 80],
[82, 89, 61],
[94, 67, 88]])
```

```
In [5]: gArray[0,2]
Out[5]: 60

In [7]: gArray[2,:]
Out[7]: array([99, 67, 84])

In [8]: gArray[:, 0]
Out[8]: array([79, 95, 99, 76, 91, 70, 88,
67, 82, 94])

In [9]: gArray[:3, :2]
Out[9]:
array([[79, 95],
[95, 60],
[99, 67]])
```

ndarray

- ndarray is used for storage of homogeneous data
  - i.e., all elements must be the same type
- Every array must have a shape
- And a dtype
- Supports convenient slicing, indexing and efficient vectorized computation
  - Avoid for loops, and much more efficient

```
In [15]: type(gArray)
Out[15]: numpy.ndarray

In [16]: gArray.ndim
Out[16]: 2

In [17]: gArray.shape
Out[17]: (10, 3)

In [18]: gArray.dtype
Out[18]: dtype('int32')
```

Creating ndarrays

- np.array
- np.zeros
- np.ones
- np.eye
- np.arrange
- np.random

```
In [65]: np.array([[0,1,2],[2,3,4]])
Out[65]:
array([[0, 1, 2],
[2, 3, 4]])

In [66]: np.zeros((2,3))
Out[66]:
array([[ 0., 0., 0.],
[ 0., 0., 0.]])

In [67]: np.ones((2,3))
Out[67]:
array([[ 1., 1., 1.],
[ 1., 1., 1.]])
```

```
In [69]: np.eye(3)
Out[69]:
array([[ 1., 0., 0.],
[ 0., 1., 0.],
[ 0., 0., 1.]])

In [70]: np.arange(0, 10, 2)
Out[70]: array([0, 2, 4, 6, 8])
```

```
In [295]: np.random.randint(0, 10, (3,3))
Out[295]:
array([[8, 7, 6],
[0, 8, 9],
[9, 0, 4]])
```

Numpy data types

- int8, int16, int32, int64
- float16, float32, float64, float128
- bool
- object
- String
- Unicode

64 bits

- gArray.astype

```
In [34]: gArray.astype(float64)
Out[34]:
array([[ 79., 95., 60.],
[ 95., 60., 61.],
...,
[ 82., 89., 61.],
[ 94., 67., 88.]])
```

```
In [79]: num_string = array(['1.0', '2.05', '3'])

In [81]: num_string
Out[81]:
array(['1.0', '2.05', '3'],
dtype='<U4')

In [82]: num_string.astype(float)
Out[82]: array([ 1. , 2.05, 3. ])
```

Array operations

- Between arrays and scalars
- Between equal-sized arrays: elementwise operation

```
In [94]: arr * arr
Out[94]:
array([[ 0, 1, 4],
[ 9, 16, 25]])

In [95]: arr / (arr+1)
Out[95]:
array([[ 0. , 0.5 , 0.66666667],
[ 0.75 , 0.8 , 0.83333333]])
```

```
In [87]: arr = array([[0,1,2],[3,4,5]])

In [88]: arr * 2
Out[88]:
array([[ 0, 2, 4],
[ 6, 8, 10]])

In [90]: arr ** 2
Out[90]:
array([[ 0, 1, 4],
[ 9, 16, 25]])

In [91]: 2 ** arr
Out[91]:
array([[ 1, 2, 4],
[ 8, 16, 32]], dtype=int32)
```

Speed difference between for loop and vectorized computation

```
In [118]: a = np.random.rand(1000000,1)
    ...: %timeit a**2
    ...: %timeit [a[i]**2 for i in range(1000000)]
100 loops, best of 3: 4.02 ms per loop
1 loop, best of 3: 1.25 s per loop
```

Vectorization is more than 300 times faster!

```
In [151]: timeit map(lambda x: x**2, a)
1000000 loops, best of 3: 270 ns per loop
```

map appears to be very fast, but it is just because it is lazy – actual calculation has not been done yet.

```
def mySum(inputList):
    s = 0
    for i in range(len(inputList)):
        s += inputList[i]
    return s
```

```
In [148]: timeit mySum(a)
1 loop, best of 3: 605 ms per loop

In [149]: timeit np.sum(a)
1000 loops, best of 3: 1.15 ms per loop

In [150]: timeit reduce(lambda x, y: x+y, a)
1 loop, best of 3: 791 ms per loop
```

Vectorization is 500 times faster than for loop.
Reduce is even slower than for loop here.

Array indexing and slicing

- Somewhat similar to python list, but much more flexible

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

```
In [153]: gArray[0]
Out[153]: array([79, 95, 60])

In [154]: gArray[1:3]
Out[154]:
array([[95, 60, 61],
       [99, 67, 84]])

In [155]: gArray[0][2]
Out[155]: 60

In [156]: gArray[0,2]
Out[156]: 60
```

```
In [157]: gArray[:, 2]
Out[157]: array([60, 61, 84, 97, 98, 96,
76, 80, 61, 88])
```

Array indexing and slicing (cont'd)

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

```
In [160]: gArray[:2, [0, 2]]
Out[160]:
array([[79, 60],
       [95, 61]])
```

```
In [175]: gArray[[0, 2], :]
Out[175]:
array([[79, 95, 60],
       [99, 67, 84]])
```

```
In [177]: gArray[[0, 2], [0, 1, 2]]
Traceback (most recent call last):
...
IndexError: shape mismatch: ...

In [178]: gArray[[0, 2], [0, 2]]
Out[178]: array([79, 84])
```

list

```
In [200]: gArray[[0,2]][:,[0,2]]
Out[200]:
array([[79, 60],
       [99, 84]])
```

```
In [272]: gArray[np.ix_([0, 2], [0, 2])]
Out[272]:
array([[79, 60],
       [99, 84]])
```

Array slices are views

```
In [202]: gArray[0,:]=100

In [203]: gArray
Out[203]:
array([[100, 100, 100],
       [ 95,  60,  61],
       [ 99,  67,  84],
       ...,
       [ 67,  73,  80],
       [ 82,  89,  61],
       [ 94,  67,  88]])
```

```
In [254]: arr2 = gArray.copy()

In [255]: arr2 is gArray
Out[255]: False

In [258]: arr2[1,:]=100

In [260]: gArray[1,:]
Out[260]: array([95, 60, 61])
```

Use .copy() to make a copy of an array explicitly.

Boolean indexing

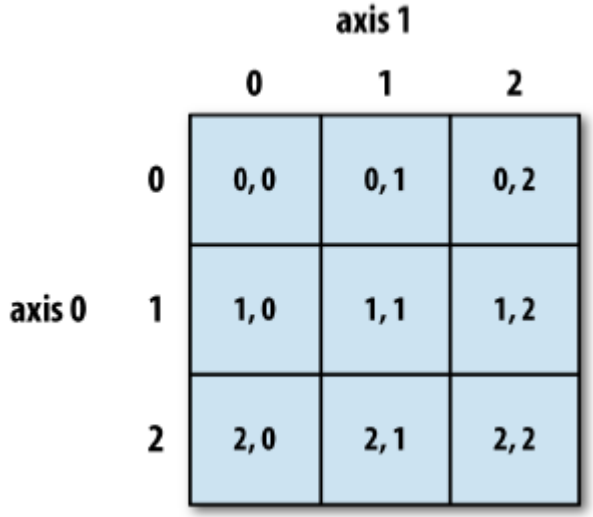
```
# select record for female students
In [262]: female = [ True, False,
True,  True, False,  True, False,
False, False, False]

In [263]: gArray[female, :]
Out[263]:
array([[100, 100, 100],
       [ 99,  67,  84],
       [ 76,  76,  97],
       [ 70,  69,  96]])
```

```
# select record for those who had
# <= 70 in final
In [265]: gArray[gArray[:,
2]<70,:]
Out[265]:
array([[95, 60, 61],
       [82, 89, 61]])

# anything < 70 is changed to 70
In [267]: gArray[gArray < 70] = 70

In [268]: gArray
Out[268]:
array([[100, 100, 100],
       [ 95,  70,  70],
       [ 99,  70,  84],
       ...,
       [ 70,  73,  80],
       [ 82,  89,  70],
       [ 94,  70,  88]])
```

Reshaping and transposing

```
In [280]: In [77]:
np.arange(6).reshape((2,3))
Out[280]:
array([[0, 1, 2],
       [3, 4, 5]])
```

```
In [281]: In [77]:
np.arange(6).reshape((2,3), order='F')
Out[281]:
array([[0, 2, 4],
       [1, 3, 5]])
```




```
In [290]:
np.arange(6).reshape(2,3).T
Out[290]:
array([[0, 3],
       [1, 4],
       [2, 5]])
```

Fast element-wise functions

Table 4-3. Unary ufuncs

Function	Description
`abs, fabs`	Compute the absolute value element-wise for integer, floating point, or complex values. Use `fabs` as a faster alternative for non-complex-valued data
`sqrt`	Compute the square root of each element. Equivalent to `arr ** 0.5`
`square`	Compute the square of each element. Equivalent to `arr ** 2`
`exp`	Compute the exponent e^x of each element
`log, log10, log2, log1p`	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
`sign`	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
`ceil`	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
`floor`	Compute the floor of each element, i.e. the largest integer less than or equal to each element
`rint`	Round elements to the nearest integer, preserving the `dtype`
`modf`	Return fractional and integral parts of array as separate array
`isnan`	Return boolean array indicating whether each value is NaN (Not a Number)
`isfinite, isinf`	Return boolean array indicating whether each element is finite (non-`inf`, non-NaN) or infinite, respectively
`cos, cosh, sin, sinh, tan, tanh`	Regular and hyperbolic trigonometric functions
`arccos, arccosh, arcsin, arcsinh, arctan, arctanh`	Inverse trigonometric functions
`logical_not`	Compute truth value of `not x` element-wise. Equivalent to `-arr`.

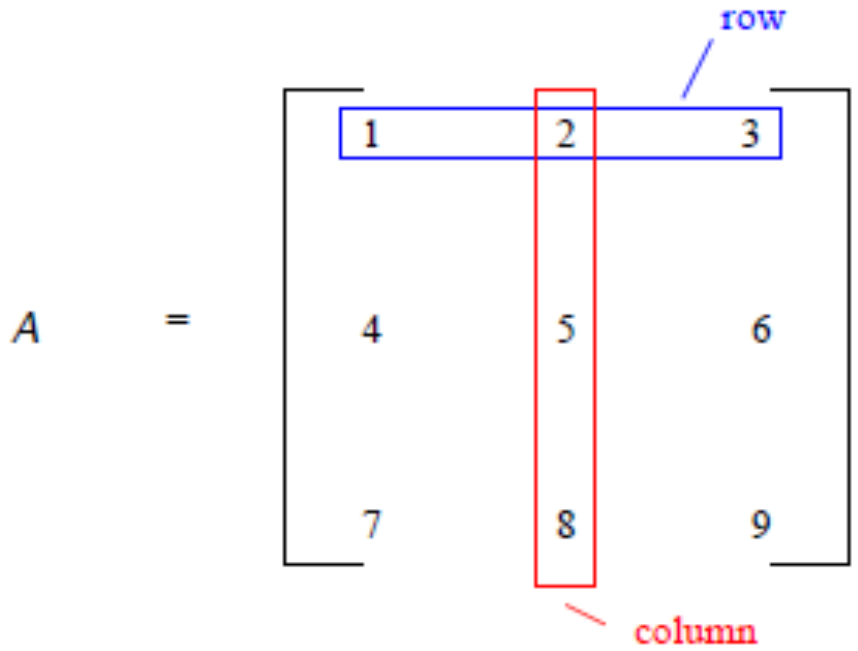
Table 4-4. Binary universal functions

Function	Description
`add`	Add corresponding elements in arrays
`subtract`	Subtract elements in second array from first array
`multiply`	Multiply array elements
`divide, floor_divide`	Divide or floor divide (truncating the remainder)
`power`	Raise elements in first array to powers indicated in second array
`maximum, fmax`	Element-wise maximum. `fmax` ignores NaN
`minimum, fmin`	Element-wise minimum. `fmin` ignores NaN
`mod`	Element-wise modulus (remainder of division)
`copysign`	Copy sign of values in second argument to values in first argument
`greater, greater_equal, less, less_equal, equal, not_equal`	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators `>, >=, <, <=, ==, !=`
`logical_and, logical_or, logical_xor`	Compute element-wise truth value of logical operation. Equivalent to infix operators `&, |, ^`

https://docs.scipy.org/doc/numpy/reference/

Matrix

- A matrix is a rectangular array of numbers organized in rows and columns
- If a matrix A has m rows and n columns, then we say that A is an m x n matrix



Matrix

- In general, a matrix A of the order $m \times n$ means:
- $$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}$$
- a_{ij} is the element of A in row i and column j.
- **row** i is the elements $a_{i1}a_{i2}\cdots a_{in}$.
- **column** j is the elements $a_{1j}a_{2j}\cdots a_{mj}$.

Vectors

- If a matrix has only one row, then it is a **row vector**.
  - Example: $\begin{pmatrix} 1 & 10 & 11 & 12 & 7 \end{pmatrix}$
- If a matrix has only one column, then it is a **column vector**.
  - Example: $\begin{pmatrix} 7 \\ -2 \\ 5 \\ 11 \end{pmatrix}$

Identity matrix

- The **identity matrix** is a square matrix that has 1s on the main diagonal and 0s everywhere else.
- An identity matrix of order $n \times n$ is denoted I_n.
- Example: $I_4 = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

Diagonal matrix

- A **diagonal matrix** is a square matrix such that every element that is not on the main diagonal is 0 (elements on the main diagonal can be 0 or non-zero).
- An $n \times n$ diagonal matrix is denoted by D_n.
- Example: $D_3 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 7 \end{pmatrix}$
- Example: $D_3 = \begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 5 \end{pmatrix}$

Dot product

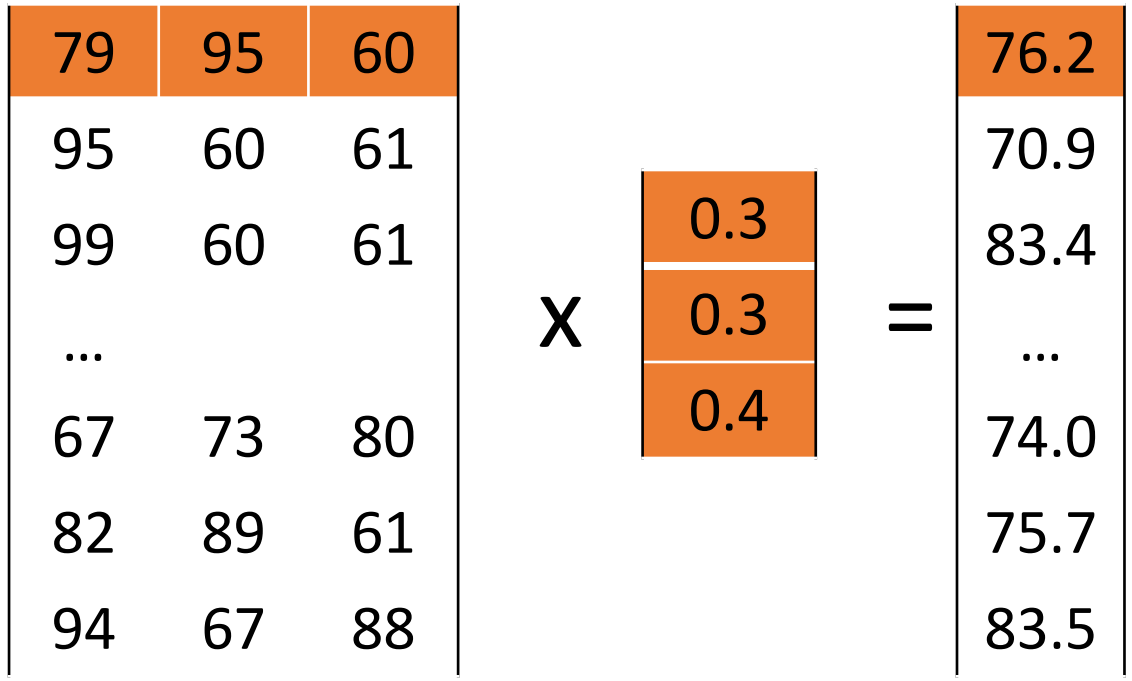
- A **dot product** is a multiplication of a row vector of order $1 \times n$ with a column vector of order $n \times 1$. The result is a scalar.
- It is obtained by multiplying the ith element of the row vector with the ith element of the row vector and then summing these products.
- $A = \begin{pmatrix} a_1 a_2 \cdots a_n \end{pmatrix}, B = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}$
- $A \cdot B = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$

```
In [303]: a = b = np.arange(5)
In [305]: a
Out[305]: array([0, 1, 2, 3, 4])
In [306]: b
Out[306]: array([0, 1, 2, 3, 4])
In [307]: a.dot(b)
Out[307]: 30
```

Matrix multiplication

- Suppose A is an $m \times p$ matrix and B is a $p \times n$ matrix (note the number of *columns* of A is the same as the number of *rows* of B). Then the **matrix multiplication** $A \cdot B$ is defined.
- The result is an $m \times n$ matrix (resulting matrix has the same number of rows as A and the same number of columns as B).
- The (i, j)th entry of the resulting matrix is the dot product of row i of A and column j of B.
- Example:
  - $A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \end{pmatrix}, B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix}$
  - Multiplication is defined because the number of columns of A is the same as the number of rows of B.
  - Result will be a 2×2 matrix.
  - Entry in position $(2, 1)$ in the resulting matrix will be the dot product of the 2nd row in A with the 1st column of B: $a_{21}b_{11} + a_{22}b_{21} + a_{23}b_{31} + a_{24}b_{41}$.

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```




```
In [321]: gArray.dot([0.3, 0.3, 0.4])
Out[321]: array([ 76.2,  70.9,  83.4,
84.4,  91.7,  80.1,  76.3,  74. ,
75.7,  83.5])
```

What is being calculated here?

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

What are we doing here?

```
In [329]: scaling = [1.1, 1.05, 1.03]

In [330]: diag(scaling)
Out[330]:
array([[ 1.1 ,  0.  ,  0.  ],
       [ 0.  ,  1.05,  0.  ],
       [ 0.  ,  0.  ,  1.03]])

In [331]: gArray.dot(diag(scaling))
Out[331]:
array([[  86.9 ,   99.75,   61.8 ],
       [ 104.5 ,   63.  ,   62.83],
       [ 108.9 ,   70.35,   86.52],
       ...,
       [  73.7 ,   76.65,   82.4 ],
       [  90.2 ,   93.45,   62.83],
       [ 103.4 ,   70.35,   90.64]])
```

```
In [152]: gArray
Out[152]:
array([[79, 95, 60],
       [95, 60, 61],
       [99, 67, 84],
       ...,
       [67, 73, 80],
       [82, 89, 61],
       [94, 67, 88]])
```

What are we doing here?

```
In [337]: gArray.max(axis=0)
Out[337]: array([99, 95, 98])

In [338]: maxInExam = gArray.max(axis=0)

In [339]:
gArray.dot(diag(100/maxInExam)).round()
Out[339]:
array([[  80.,  100.,   61.],
       [  96.,   63.,   62.],
       [ 100.,   71.,   86.],
       ...,
       [  68.,   77.,   82.],
       [  83.,   94.,   62.],
       [  95.,   71.,   90.]])
```

Speed difference between for loop and matrix multiplication

```
In [361]: maxInCol = a.max(axis=0)
In [362]: maxInCol.shape
Out[362]: (100,)

In [363]: timeit [[a[j,i]*100/maxInCol[i]
                   for i in range(100)]
                       for j in range(10000)]
1 loop, best of 3: 673 ms per loop
```

```
In [355]: a = rand(10000, 100)

In [356]: timeit a.dot(100/a.max(0))
100 loops, best of 3: 1.77 ms per loop

In [357]: timeit [a[:,i]*100/max(a[:,i]) for i in range(100)]
10 loops, best of 3: 72.9 ms per loop

In [358]: timeit [[a[j,i]*100/max(a[:,i]) for i in range(100)]
                                           for j in range(10000)]

Ctrl-C
```

Table 4-5. Basic array statistical methods

Method	Description
`sum`	Sum of all the elements in the array or along an axis. Zero-length arrays have sum 0.
`mean`	Arithmetic mean. Zero-length arrays have NaN mean.
`std, var`	Standard deviation and variance, respectively, with optional degrees of freedom adjustment (default denominator n).
`min, max`	Minimum and maximum.
`argmin, argmax`	Indices of minimum and maximum elements, respectively.
`cumsum`	Cumulative sum of elements starting from 0
`cumprod`	Cumulative product of elements starting from 1

```
In [394]: a=randint(0, 5, size=(3,3))
In [395]: a
Out[395]:
array([[4, 1, 2],
       [2, 2, 1],
       [1, 1, 4]])

In [396]: a.sum()
Out[396]: 18
```

```
In [397]: a.sum(axis=0)
Out[397]: array([7, 4, 7])

In [398]: a.sum(1)
Out[398]: array([7, 5, 6])

In [405]: (a > 2).any(1)
Out[405]: array([ True, False,  True],
dtype=bool)
```

numpy.sort()

```
In [407]: a.sort()

In [408]: a
Out[408]:
array([[1, 2, 4],
       [1, 2, 2],
       [1, 1, 4]])
```

```
In [410]: a.sort(0)

In [411]: a
Out[411]:
array([[1, 1, 2],
       [1, 2, 4],
       [1, 2, 4]])
```

Adjacency matrix for a graph

```
In [603]: n=friends.max()
     ...: frdGraph = zeros((n+1,n+1))
     ...: frdGraph[friends[:,0],friends[:,1]]=1
     ...: frdGraph[friends[:,1],friends[:,0]]=1
     ...: imshow(frdGraph)
     ...: xticks(range(9))
```

```
In [604]: friends
Out[604]:
array([[0, 2],
       [0, 6],
       [1, 3],
       ...,
       [4, 7],
       [5, 8],
       [6, 7]],
dtype=int64)
```

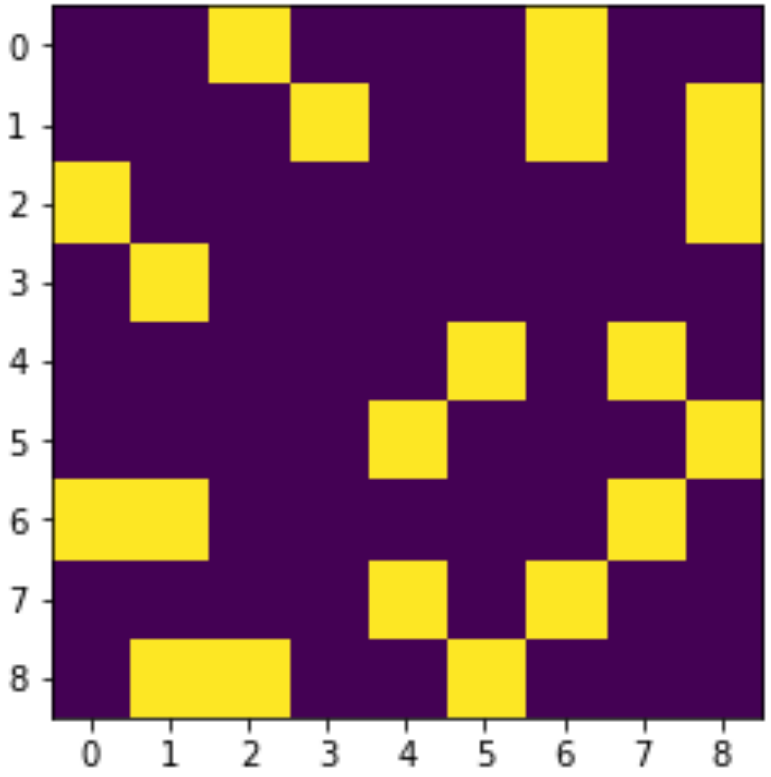


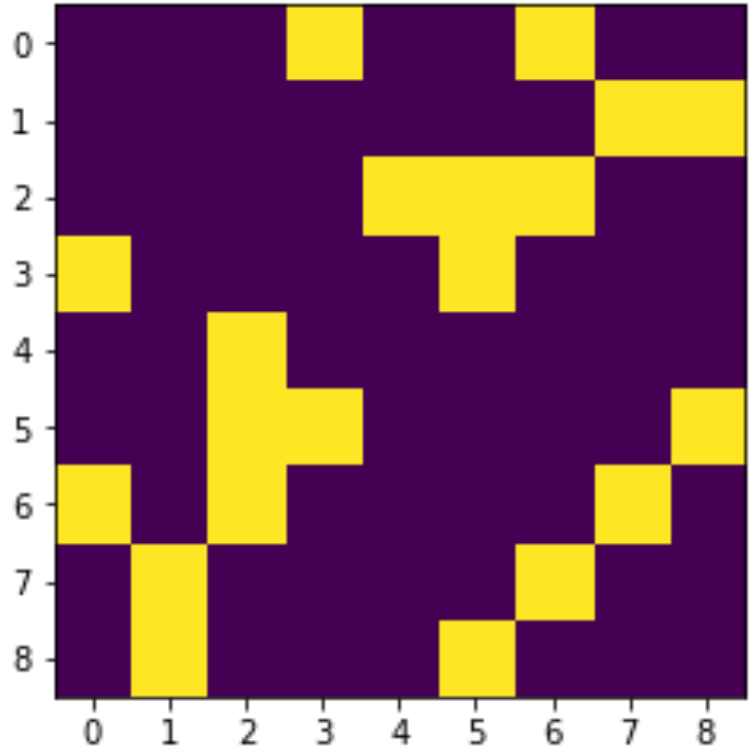
Table 4-6. Array set operations

Method	Description
`unique(x)`	Compute the sorted, unique elements in x
`intersect1d(x, y)`	Compute the sorted, common elements in x and y
`union1d(x, y)`	Compute the sorted union of elements
`in1d(x, y)`	Compute a boolean array indicating whether each element of x is contained in y
`setdiff1d(x, y)`	Set difference, elements in x that are not in y
`setxor1d(x, y)`	Set symmetric differences; elements that are in either of the arrays, but not both

```
In [654]: edgeList
Out[654]:
array([['Amy', 'Frank'],
['Amy', 'Katy'],
['Emma', 'James'],
...,
['Cindy', 'Rose'],
['Tim', 'John'],
['Katy', 'Rose']],
dtype='<U5')
```

```
In [676]: names, indices = unique(edgeList, return_inverse=True)
In [677]: names
Out[677]:
array(['Amy', 'Cindy', 'Emma', 'Frank', 'James', 'John', 'Katy', 'Rose', 'Tim'],  dtype='<U5')
In [678]: indices
Out[678]: array([0, 3, 0, ..., 5, 6, 7], dtype=int64)
In [715]: reshape(indices,(-1, 2)).T
Out[715]:
array([[0, 0, 2, ..., 1, 8, 6],
       [3, 6, 4, ..., 7, 5, 7]], dtype=int64)
```

```
In [654]: edgeList
Out[654]:
array([['Amy', 'Frank'],
['Amy', 'Katy'],
['Emma', 'James'],
...,
['Cindy', 'Rose'],
['Tim', 'John'],
['Katy', 'Rose']],
dtype='<U5')
```




```
In [676]: names, indices = unique(edgeList, return_inverse=True)
In [677]: names
Out[677]:
array(['Amy', 'Cindy', 'Emma', 'Frank', 'James', 'John', 'Katy', 'Rose', 'Tim'],  dtype='<U5')
In [710]: n = indices.max()
    ...: frdGraph2 = zeros((n+1,n+1))
    ...: frdGraph2[indices[::2], indices[1::2]] = 1
    ...: frdGraph2[indices[1::2], indices[::2]] = 1
    ...: imshow(frdGraph2); xticks(range(n+1))
```

Sparse matrix support

- In scipy.sparse
- Necessary for larger sparse graphs (e.g. social networks)
- Most real world networks are sparse
- Memory efficiency is crucial for applications

https://docs.scipy.org/doc/scipy/reference/index.html

Additional materials

- https://www.geeksforgeeks.org/numpy-in-python-set-1-introduction/
- https://numpy.org/learn/