Lecture 13, 26 September 2023

Using numpy

- Arrays and lists
- Arrays are "homogenous" with regular structure
- Lists are flexible

Load numpy

In [1]:
```
import numpy as np
```

Constructing arrays

`np.array()` constructs an array from an input sequence

- Sequence can be a list, tuple, output of a `range()` command ...
- Size of the array is fixed by the sequence
- Underlying type is also fixed

In [2]:
```
b = np.array(range(10))
b
```

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

Use nested sequences to produce multi-dimensional arrays

- A 2d array is an array of 1d arrays
- Note: can mix and match notation for sequences, but dimensions much match

In [3]:
```
c = np.array([(0,1,0),[2,3,2]])
c
```

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

In [4]:
```
d = np.array([(0,1,0),range(3)])
d
```

Out[4]:
```
array([[0, 1, 0],
       [0, 1, 2]])
```

- A 3d array is an array of 2d arrays

In [5]:
```
d = np.array([[(0,1,0),[2,3,2]],[[4,5,4],[6,7,6]]])
d
```

Out[5]:
```
array([[[0, 1, 0],
        [2, 3, 2]],

       [[4, 5, 4],
        [6, 7, 6]]])
```

Pointwise scalar operations --- broadcasting

In [6]:
```
a = np.arange(10)  # arange(n) is same as array(range(n))
a
```

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

In [7]:
```
a**3  # Replace each element by its cube
```

Out[7]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

In [8]:
```
a+3   # Add 3 to each element
```

Out[8]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [9]:
```
3*a   # Multiply each element by 3
```

Out[9]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

In [10]:
```
3+a
```

Out[10]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [11]:
```
a-3, 3-a
```

Out[11]:
```
(array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6]),
 array([ 3,  2,  1,  0, -1, -2, -3, -4, -5, -6]))
```

In [12]:
```
3**a
```

Out[12]:
```
array([    1,     3,     9,    27,    81,   243,   729,  2187,  6561,
       19683])
```

Stacking arrays

In [13]:
```
a = np.floor(10*np.random.random((2,2)))
b = np.floor(10*np.random.random((2,2)))
print(a)
print(b)
```

```
[[2. 8.]
 [6. 4.]]
[[8. 9.]
 [5. 4.]]
```

`vstack` stacks a sequence of arrays vertically -- should have same number of columns

In [14]:
```
np.vstack((a,b))
```

Out[14]:
```
array([[2., 8.],
       [6., 4.],
       [8., 9.],
       [5., 4.]])
```

In [15]:
```
c = np.floor(10*np.random.random((3,3)))
c
```

Out[15]:
```
array([[6., 6., 5.],
       [7., 1., 6.],
       [6., 4., 3.]])
```

In [16]:
```
np.vstack((a,c))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[16], line 1
----> 1 np.vstack((a,c))

File ~/miniconda3/lib/python3.11/site-packages/numpy/core/shape_base.py:289, in vstack(tup, dtype, casti
ng)
    287 if not isinstance(arrs, list):
    288     arrs = [arrs]
--> 289 return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)

ValueError: all the input array dimensions except for the concatenation axis must match exactly, but alo
ng dimension 1, the array at index 0 has size 2 and the array at index 1 has size 3
```

In [17]:
```
d = np.floor(10*np.random.random((3,2)))
d
```

Out[17]:
```
array([[4., 2.],
       [6., 7.],
       [2., 8.]])
```

In [18]:
```
np.vstack((a,d))
```

Out[18]:
```
array([[2., 8.],
       [6., 4.],
       [4., 2.],
       [6., 7.],
       [2., 8.]])
```

Can stack any length sequence of arrays, not just two arrays

In [19]:
```
np.vstack([a,b,d])
```

Out[19]:
```
array([[2., 8.],
       [6., 4.],
       [8., 9.],
       [5., 4.],
       [4., 2.],
       [6., 7.],
       [2., 8.]])
```

Likewise, `hstack` stacks horizontally, number of rows must match

In [20]:
```
np.hstack((a,b))
```

Out[20]:
```
array([[2., 8., 8., 9.],
       [6., 4., 5., 4.]])
```

In [21]:
```
np.hstack((b,c))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[21], line 1
----> 1 np.hstack((b,c))

File ~/miniconda3/lib/python3.11/site-packages/numpy/core/shape_base.py:359, in hstack(tup, dtype, casti
ng)
    357     return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
    358 else:
--> 359     return _nx.concatenate(arrs, 1, dtype=dtype, casting=casting)

ValueError: all the input array dimensions except for the concatenation axis must match exactly, but alo
ng dimension 0, the array at index 0 has size 2 and the array at index 1 has size 3
```

In [22]:
```
e = np.floor(10*np.random.random((2,3)))
e
```

Out[22]:
```
array([[9., 3., 0.],
       [6., 6., 3.]])
```

In [23]:
```
np.hstack((a,b,e))
```

Out[23]:
```
array([[2., 8., 8., 9., 9., 3., 0.],
       [6., 4., 5., 4., 6., 6., 3.]])
```

Splitting arrays

In [24]:
```
a = np.floor(10*np.random.random((2,12)))
a
```

Out[24]:
```
array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
       [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]])
```

- `hsplit(A,n)` splits array `A` into `n` equal parts horizontally
- `n` must be a divisor of the number of columns in `A`

In [25]:
```
np.hsplit(a,3)
```

Out[25]:
```
[array([[3., 4., 3., 3.],
        [9., 3., 4., 8.]]),
 array([[4., 2., 5., 4.],
        [9., 1., 6., 3.]]),
 array([[3., 6., 6., 7.],
        [7., 8., 9., 0.]])]
```

In [26]:
```
np.hsplit(a,6)
```

Out[26]:
```
[array([[3., 4.],
        [9., 3.]]),
 array([[3., 3.],
        [4., 8.]]),
 array([[4., 2.],
        [9., 1.]]),
 array([[5., 4.],
        [6., 3.]]),
 array([[3., 6.],
        [7., 8.]]),
 array([[6., 7.],
        [9., 0.]])]
```

In [27]:
```
np.hsplit(a,5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[27], line 1
----> 1 np.hsplit(a,5)

File ~/miniconda3/lib/python3.11/site-packages/numpy/lib/shape_base.py:938, in hsplit(ary, indices_or_se
ctions)
    936     raise ValueError('hsplit only works on arrays of 1 or more dimensions')
    937 if ary.ndim > 1:
--> 938     return split(ary, indices_or_sections, 1)
    939 else:
    940     return split(ary, indices_or_sections, 0)

File ~/miniconda3/lib/python3.11/site-packages/numpy/lib/shape_base.py:864, in split(ary, indices_or_sec
tions, axis)
    862     N = ary.shape[axis]
    863     if N % sections:
--> 864         raise ValueError(
    865             'array split does not result in an equal division') from None
    866 return array_split(ary, indices_or_sections, axis)

ValueError: array split does not result in an equal division
```

- Can also specify where to split as a list of columns
- `hsplit(A,[c1,c2,..,ck])` will split like `A[:c1]`, `A[c1:c2]`,..., `A[ck:]`

In [28]:
```
np.hsplit(a,(2,5,7)) # a[:2], a[2:5], a[5:7], a[7:]
```

Out[28]:
```
[array([[3., 4.],
        [9., 3.]]),
 array([[3., 3., 4.],
        [4., 8., 9.]]),
 array([[2., 5.],
        [1., 6.]]),
 array([[4., 3., 6., 6., 7.],
        [3., 7., 8., 9., 0.]])]
```

- Similarly, `vsplit` for vertical split

In [29]:
```
np.vsplit(a,2) # Split a vertically
```

Out[29]:
```
[array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.]]),
 array([[9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]])]
```

In [30]:
```
np.split(a,2) # behaves like vsplit
```

Out[30]:
```
[array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.]]),
 array([[9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]])]
```

Copy and view

In [31]:
```
c = a.copy()  # Creates a disjoint copy of the array
d = a.view()  # Creates another link to the same array
e = a         # Aliases e to point to same array as a
```

In [32]:
```
a, c, d, e
```

Out[32]:
```
(array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]),
 array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]),
 array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]),
 array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]))
```

- Updating `c` has no effect on the others since it is a disjoint copy

In [33]:
```
c[0,4] = 88
a, c, d, e
```

Out[33]:
```
(array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]),
 array([[3., 4., 3., 3., 4., 2., 5., 4., 3., 6., 6., 7.],
        [9., 3., 4., 8., 9., 1., 6., 3., 7., 8., 9., 0.]]))
```

- Updating `d` will indirectly update `a` and `e`, but not `c`

In [34]:
```
d[0,5] = 66
a, c, d, e
```

Out[34]:
```
(array([[ 3.,  4.,  3.,  3.,  4., 66.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]))
```

- Likewise, updating `e` updates `a` and `d`

In [35]:
```
e[0,6] = 77
a, c, d, e
```

Out[35]:
```
(array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]))
```

In [36]:
```
for i in a.flat:
    print(i,end=" ")
```

```
3.0 4.0 3.0 3.0 4.0 66.0 77.0 4.0 3.0 6.0 6.0 7.0 9.0 3.0 4.0 8.0 9.0 1.0 6.0 3.0 7.0 8.0 9.0 0.0
```

- `base` tells us if an array shares its storage with another array
- For the original array, `base` is `None`
- For a view, the `base` points to the "parent" array

In [37]:
```
a.base, c.base, d.base, e.base
```

Out[37]:
```
(None,
 None,
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 None)
```

In [38]:
```
d.base is a
```

Out[38]:
```
True
```

Reshaping arrays

- Can change the *shape* of an array if the dimensions match
- View is not affected by this

In [39]:
```
a.shape
```

Out[39]:
```
(2, 12)
```

In [40]:
```
a.shape = 4,6
a,c,d,e
```

Out[40]:
```
(array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [ 6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [ 6.,  3.,  7.,  8.,  9.,  0.]]))
```

In [41]:
```
d.shape = 3,8
a,c,d,e
```

Out[41]:
```
(array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [ 6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.],
        [ 3.,  6.,  6.,  7.,  9.,  3.,  4.,  8.],
        [ 9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [ 6.,  3.,  7.,  8.,  9.,  0.]]))
```

In [42]:
```
a[3,0] = 99
a,c,d,e
```

Out[42]:
```
(array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [99.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.],
        [ 3.,  6.,  6.,  7.,  9.,  3.,  4.,  8.],
        [ 9.,  1., 99.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66.],
        [77.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.],
        [99.,  3.,  7.,  8.,  9.,  0.]]))
```

In [43]:
```
e.shape = 8,3
a,c,d,e
```

Out[43]:
```
(array([[ 3.,  4.,  3.],
        [ 3.,  4., 66.],
        [77.,  4.,  3.],
        [ 6.,  6.,  7.],
        [ 9.,  3.,  4.],
        [ 8.,  9.,  1.],
        [99.,  3.,  7.],
        [ 8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3., 88.,  2.,  5.,  4.,  3.,  6.,  6.,  7.],
        [ 9.,  3.,  4.,  8.,  9.,  1.,  6.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.,  3.,  4., 66., 77.,  4.],
        [ 3.,  6.,  6.,  7.,  9.,  3.,  4.,  8.],
        [ 9.,  1., 99.,  3.,  7.,  8.,  9.,  0.]]),
 array([[ 3.,  4.,  3.],
        [ 3.,  4., 66.],
        [77.,  4.,  3.],
        [ 6.,  6.,  7.],
        [ 9.,  3.,  4.],
        [ 8.,  9.,  1.],
        [99.,  3.,  7.],
        [ 8.,  9.,  0.]]))
```

Matrix operations

In [44]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
```

In [45]:
```
a,b
```

Out[45]:
```
(array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]]))
```

- Pointwise addition and multiplication

In [46]:
```
a+b, a*b
```

Out[46]:
```
(array([[ 6,  8],
        [10, 12]]),
 array([[ 5, 12],
        [21, 32]]))
```

- Matrix multiplication

In [47]:
```
np.matmul(a,b)
```

Out[47]:
```
array([[19, 22],
       [43, 50]])
```

- Transpose and inverse

In [48]:
```
a.T
```

Out[48]:
```
array([[1, 3],
       [2, 4]])
```

In [49]:
```
np.linalg.inv(a)
```

Out[49]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

- AA^{-1} should give the identity matrix
- Note the small imprecision due to round off error

In [50]:
```
np.matmul(a,np.linalg.inv(a))
```

Out[50]:
```
array([[1.00000000e+00, 1.11022302e-16],
       [0.00000000e+00, 1.00000000e+00]])
```

- Fit a function f to a set of data points $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$
- Compute *mean square error (MSE)*

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(f(x_i) - y_i)^2$$

In [51]:
```
predictions = np.array([1.2,2.3,3.1])
values = np.array([1,2.5,3])
```

In [52]:
```
n = len(predictions)
mse = 1/n * np.sum(np.square(predictions-values))
mse
```

Out[52]:
```
0.03000000000000002
```

Axes

In [53]:
```
a = np.random.random((4,3))*10
a
```

Out[53]:
```
array([[2.35162434, 3.17561428, 6.01028914],
       [2.47564989, 1.14038437, 9.48830529],
       [2.92970998, 0.93745306, 8.59080686],
       [2.1795133 , 8.33338406, 0.99432175]])
```

In [54]:
```
np.sum(a)
```

Out[54]:
```
48.60705631699458
```

In [55]:
```
np.sum(a,axis=0)
```

Out[55]:
```
array([ 9.93649751, 13.58683577, 25.08372305])
```

In [56]:
```
np.sum(a,axis=1)
```

Out[56]:
```
array([11.53752776, 13.10433955, 12.45796989, 11.50721911])
```

In [57]:
```
b = np.random.random((4,3))*10
b
```

Out[57]:
```
array([[9.32467504e+00, 5.12338586e+00, 3.94472072e-03],
       [5.47626949e-01, 8.13627504e+00, 9.46387236e+00],
       [5.82511033e+00, 6.77309104e-01, 9.15675693e+00],
       [6.06770698e+00, 8.03649880e+00, 7.67869295e+00]])
```

In [58]:
```
np.vstack((a,b)), np.hstack((a,b))
```

Out[58]:
```
(array([[2.35162434e+00, 3.17561428e+00, 6.01028914e+00],
        [2.47564989e+00, 1.14038437e+00, 9.48830529e+00],
        [2.92970998e+00, 9.37453055e-01, 8.59080686e+00],
        [2.17951330e+00, 8.33338406e+00, 9.94321752e-01],
        [9.32467504e+00, 5.12338586e+00, 3.94472072e-03],
        [5.47626949e-01, 8.13627504e+00, 9.46387236e+00],
        [5.82511033e+00, 6.77309104e-01, 9.15675693e+00],
        [6.06770698e+00, 8.03649880e+00, 7.67869295e+00]]),
 array([[2.35162434e+00, 3.17561428e+00, 6.01028914e+00, 9.32467504e+00,
         5.12338586e+00, 3.94472072e-03],
        [2.47564989e+00, 1.14038437e+00, 9.48830529e+00, 5.47626949e-01,
         8.13627504e+00, 9.46387236e+00],
        [2.92970998e+00, 9.37453055e-01, 8.59080686e+00, 5.82511033e+00,
         6.77309104e-01, 9.15675693e+00],
        [2.17951330e+00, 8.33338406e+00, 9.94321752e-01, 6.06770698e+00,
         8.03649880e+00, 7.67869295e+00]]))
```

In [59]:
```
np.concatenate((a,b))
```

Out[59]:
```
array([[2.35162434e+00, 3.17561428e+00, 6.01028914e+00],
       [2.47564989e+00, 1.14038437e+00, 9.48830529e+00],
       [2.92970998e+00, 9.37453055e-01, 8.59080686e+00],
       [2.17951330e+00, 8.33338406e+00, 9.94321752e-01],
       [9.32467504e+00, 5.12338586e+00, 3.94472072e-03],
       [5.47626949e-01, 8.13627504e+00, 9.46387236e+00],
       [5.82511033e+00, 6.77309104e-01, 9.15675693e+00],
       [6.06770698e+00, 8.03649880e+00, 7.67869295e+00]])
```

In [60]:
```
np.concatenate((a,b),axis=1)
```

Out[60]:
```
array([[2.35162434e+00, 3.17561428e+00, 6.01028914e+00, 9.32467504e+00,
        5.12338586e+00, 3.94472072e-03],
       [2.47564989e+00, 1.14038437e+00, 9.48830529e+00, 5.47626949e-01,
        8.13627504e+00, 9.46387236e+00],
       [2.92970998e+00, 9.37453055e-01, 8.59080686e+00, 5.82511033e+00,
        6.77309104e-01, 9.15675693e+00],
       [2.17951330e+00, 8.33338406e+00, 9.94321752e-01, 6.06770698e+00,
        8.03649880e+00, 7.67869295e+00]])
```

In [61]:
```
c = np.random.random((1,7))
d = np.random.random((1,7))
c,d
```

Out[61]:
```
(array([[0.59120274, 0.82842833, 0.05923415, 0.35925587, 0.12135288,
         0.38540662, 0.23041868]]),
 array([[0.29055153, 0.47270183, 0.48871466, 0.0273599 , 0.52315667,
         0.71149695, 0.9304671 ]]))
```

In [62]:
```
np.concatenate((c,d),axis=1)
```

Out[62]:
```
array([[0.59120274, 0.82842833, 0.05923415, 0.35925587, 0.12135288,
        0.38540662, 0.23041868, 0.29055153, 0.47270183, 0.48871466,
        0.0273599 , 0.52315667, 0.71149695, 0.9304671 ]])
```

Broadcasting

- Array with scalar --- map operation to each array element

In [63]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Out[63]:
```
array([2., 4., 6.])
```

- Array with array/sequence of same length --- pointwise application of operation

In [64]:
```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
```

Out[64]:
```
array([2., 4., 6.])
```

- More generally, can broadcast to an array of same dimension as rightmost index

Example

- Store an $m \times n$ image as 3 layers, Red, Blue and Green
- Array dimensions are $(m, n, 3)$
- Want to scale RGB values by different amounts
- Multiply image by $(rscale, bscale, gscale)$

In [65]:
```
pic = np.random.random((4,4,3))*10
```

In [66]:
```
pic
```

Out[66]:
```
array([[[1.37164148, 5.11706695, 0.79112626],
        [5.85870596, 6.99262418, 6.82648383],
        [2.19538699, 3.69562375, 3.39149786],
        [7.58343024, 3.36452428, 8.07544361]],

       [[6.4202232 , 0.8294357 , 8.38854742],
        [1.33415726, 2.27046016, 2.3085659 ],
        [2.21025393, 6.02381327, 5.30899257],
        [5.49801637, 4.93831805, 6.81495139]],

       [[9.5682806 , 2.66767877, 9.26300957],
        [5.14001852, 9.95944033, 9.59606362],
        [6.27213828, 1.09623904, 8.63631764],
        [6.17085691, 8.11334873, 5.02499077]],

       [[0.35279941, 4.07730249, 7.7939125 ],
        [0.71975111, 9.1127831 , 1.0272418 ],
        [5.97895041, 7.53430495, 7.04543546],
        [3.89341431, 4.4844843 , 0.74430765]]])
```

In [67]:
```
pic*[1,100,1000]
```

Out[67]:
```
array([[[1.37164148e+00, 5.11706695e+02, 7.91126262e+02],
        [5.85870596e+00, 6.99262418e+02, 6.82648383e+03],
        [2.19538699e+00, 3.69562375e+02, 3.39149786e+03],
        [7.58343024e+00, 3.36452428e+02, 8.07544361e+03]],

       [[6.42022320e+00, 8.29435699e+01, 8.38854742e+03],
        [1.33415726e+00, 2.27046016e+02, 2.30856590e+03],
        [2.21025393e+00, 6.02381327e+02, 5.30899257e+03],
        [5.49801637e+00, 4.93831805e+02, 6.81495139e+03]],

       [[9.56828060e+00, 2.66767877e+02, 9.26300957e+03],
        [5.14001852e+00, 9.95944033e+02, 9.59606362e+03],
        [6.27213828e+00, 1.09623904e+02, 8.63631764e+03],
        [6.17085691e+00, 8.11334873e+02, 5.02499077e+03]],

       [[3.52799406e-01, 4.07730249e+02, 7.79391250e+03],
        [7.19751112e-01, 9.11278310e+02, 1.02724180e+03],
        [5.97895041e+00, 7.53430495e+02, 7.04543546e+03],
        [3.89341431e+00, 4.48448430e+02, 7.44307652e+02]]])
```

Broadcasting example

- Find the nearest point to a given point in a collection
- Given (x, y) and $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$, report j such that (x_j, y_j) is the closest point to (x, y)
- Distance of each point is $\sqrt{(x_i - x)^2 + (y_i - y)^2}$
- $\arg\min$ reports index where $\min$ is achieved

In [68]:
```
observation = np.array([111.0, 188.0])
codes = np.array([[132.0, 193.0],
                  [102.0, 203.0],
                  [45.0, 155.0],
                  [57.0, 173.0]])
diff = codes - observation    # the broadcast happens here
dist = np.sqrt(np.sum(diff**2,axis=1))
dist, np.argmin(dist)
```

Out[68]:
```
(array([21.58703314, 17.49285568, 73.79024326, 56.04462508]), 1)
```