Lecture 18, 16 October 2025

Using numpy

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

Load numpy

In [1]:
```
import numpy as np  # Shortcut, use np for numpy in code
```

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])
```

In [3]:
```
b.dtype # Underlying datatype of b
```

Out[3]:
```
dtype('int64')
```

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 [4]:
```
c = np.array([(0,1,0),[2,3,2]])
c
```

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

In [5]:
```
c = np.array([(0,1,0),[2,2]])
c
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[5], line 1
----> 1 c = np.array([(0,1,0),[2,2]])
      2 c

ValueError: setting an array element with a sequence. The requested array has an inhomogeneous shape after 1 dimensions. The detected shape was (2,) + inhomogeneous part.
```

In [6]:
```
d = np.array([(0,1,0),range(3)])
d
```

Out[6]:
```
array([[0, 1, 0],
       [0, 1, 2]])
```

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

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

Out[7]:
```
array([[[0, 1, 0],
        [2, 3, 2]],

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

In [8]:
```
m_zeros = np.zeros((5,3))  # np.zeros(shape), shape = (dim1,dim2,...,dimk
m_zeros
```

Out[8]:
```
array([[0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

In [9]:
```
m_zeros = np.zeros(5,3)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[9], line 1
----> 1 m_zeros = np.zeros(5,3)

TypeError: Cannot interpret '3' as a data type
```

In [11]:
```
m_zeros.dtype
```

Out[11]:
```
dtype('float64')
```

In [12]:
```
m_zero_int = np.zeros(7,dtype='int64')
m_zero_int
```

Out[12]:
```
array([0, 0, 0, 0, 0, 0, 0])
```

In [13]:
```
m_ones = np.ones((8,2,3))
m_ones
```

Out[13]:
```
array([[[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]],

       [[1., 1., 1.],
        [1., 1., 1.]]])
```

In [14]:
```
m_rand = np.random.random((4,6))
m_rand
```

Out[14]:
```
array([[0.61478095, 0.66873022, 0.95844102, 0.42150011, 0.4276577 ,
        0.4466395 ],
       [0.4091643 , 0.19277623, 0.64426873, 0.29413532, 0.3060221 ,
        0.55254819],
       [0.96782399, 0.64445448, 0.30810994, 0.22376551, 0.87498372,
        0.34494317],
       [0.69119246, 0.57612184, 0.57630797, 0.92630779, 0.64981307,
        0.38438788]])
```

In [15]:
```
m_rand_int = np.random.randint(1,100,(3,4))
m_rand_int
```

Out[15]:
```
array([[89, 84, 64, 20],
       [10, 51, 13, 23],
       [ 2, 35,  5, 34]])
```

In [16]:
```
m_normal = np.random.normal(0,1,(5,2))   # np.normal(mean, std_dev, shape
m_normal
```

Out[16]:
```
array([[ 2.25324418,  0.02261622],
       [ 0.46259818, -0.28531591],
       [-0.83403691, -2.6713428 ],
       [ 1.17590504, -1.09052901],
       [-0.98658065, -0.21246764]])
```

Broadcasting

- In simplest form, scalar operations applied pointwise

In [17]:
```
a = np.arange(10)  # arange(n) is same as array(range(n))
a
```

Out[17]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [18]:
```
a**3  # Replace each element by its cube  --- [ x**3 for x in a ]
```

Out[18]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

In [19]:
```
a+3   # Add 3 to each element
```

Out[19]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [20]:
```
3*a   # Multiply each element by 3
```

Out[20]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

In [21]:
```
3+a   # Same as a+3
```

Out[21]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [22]:
```
a-3, 3-a  # Order is important because subtraction is not commutative
```

Out[22]:
```
(array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6]),
 array([ 3,  2,  1,  0, -1, -2, -3, -4, -5, -6]))
```

In [23]:
```
3**a  # Powers of 3
```

Out[23]:
```
array([    1,     3,     9,    27,    81,   243,   729,  2187,  6561,
       19683])
```

Stacking arrays

- `np.random.random(m,n)` creates an mxn array with random numbers drawn uniformly from $[0, 1)$
- `10*np.random.random()` scales the entries by 10
- `np.floor()` removes the fractional part

In [24]:
```
a = np.floor(10*np.random.random((2,2)))
b = np.floor(10*np.random.random((3,2)))
print(a)
print(b)
```

```
[[4. 1.]
 [6. 9.]]
[[1. 5.]
 [2. 1.]
 [5. 2.]]
```

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

In [25]:
```
np.vstack((a,b))
```

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

- Cannot `vstack()` if number of columns don't match

In [26]:
```
c = np.floor(10*np.random.random((2,3)))
c
```

Out[26]:
```
array([[2., 8., 2.],
       [9., 4., 9.]])
```

In [27]:
```
np.vstack((a,c))
```

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

File ~/python-venv/lib/python3.13/site-packages/numpy/_core/shape_base.py:292, in vstack(tup, dtype, casting)
    290 if not isinstance(arrs, tuple):
    291     arrs = (arrs,)
--> 292 return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)

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

- Number of rows does not matter if columns match

In [28]:
```
d = np.floor(10*np.random.random((3,2)))
d
```

Out[28]:
```
array([[1., 2.],
       [2., 2.],
       [6., 0.]])
```

In [29]:
```
np.vstack((a,d))
```

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

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

In [30]:
```
np.vstack([a,b,d])
```

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

- Likewise, `hstack()` stacks horizontally, number of rows must match

In [31]:
```
np.hstack((a,b))
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[31], line 1
----> 1 np.hstack((a,b))

File ~/python-venv/lib/python3.13/site-packages/numpy/_core/shape_base.py:367, in hstack(tup, dtype, casting)
    365     return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
    366 else:
--> 367     return _nx.concatenate(arrs, 1, dtype=dtype, casting=casting)

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

In [32]:
```
np.hstack((b,d))
```

Out[32]:
```
array([[1., 5., 1., 2.],
       [2., 1., 2., 2.],
       [5., 2., 6., 0.]])
```

- For `hstack()`, number of columns does not matter
- Like `vstack()`, can combine a sequence of arrays

In [33]:
```
e = np.floor(10*np.random.random((3,3)))
e
```

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

In [34]:
```
np.hstack((b,d,e))
```

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

Splitting arrays

In [35]:
```
a = np.floor(10*np.random.random((2,12)))
a
```

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

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

In [36]:
```
np.hsplit(a,3)
```

Out[36]:
```
[array([[1., 1., 6., 1.],
        [0., 9., 3., 7.]]),
 array([[7., 2., 8., 1.],
        [6., 7., 0., 2.]]),
 array([[4., 6., 4., 4.],
        [0., 7., 9., 1.]])]
```

In [37]:
```
np.hsplit(a,6)
```

Out[37]:
```
[array([[1., 1.],
        [0., 9.]]),
 array([[6., 1.],
        [3., 7.]]),
 array([[7., 2.],
        [6., 7.]]),
 array([[8., 1.],
        [0., 2.]]),
 array([[4., 6.],
        [0., 7.]]),
 array([[4., 4.],
        [9., 1.]])]
```

In [38]:
```
np.hsplit(a,5)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[38], line 1
----> 1 np.hsplit(a,5)

File ~/python-venv/lib/python3.13/site-packages/numpy/lib/_shape_base_impl.py:959, in hsplit(ary, indices_or_sections)
    957     raise ValueError('hsplit only works on arrays of 1 or more dimensions')
    958 if ary.ndim > 1:
--> 959     return split(ary, indices_or_sections, 1)
    960 else:
    961     return split(ary, indices_or_sections, 0)

File ~/python-venv/lib/python3.13/site-packages/numpy/lib/_shape_base_impl.py:884, in split(ary, indices_or_sections, axis)
    882     N = ary.shape[axis]
    883     if N % sections:
--> 884         raise ValueError(
    885             'array split does not result in an equal division') from None
    886 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 as `A[:c1]`, `A[c1:c2]`,..., `A[ck:]`

In [39]:
```
np.hsplit(a,(2,5,7)) # a[:2], a[2:5], a[5:7], a[7:]
```

Out[39]:
```
[array([[1., 1.],
        [0., 9.]]),
 array([[6., 1., 7.],
        [3., 7., 6.]]),
 array([[2., 8.],
        [7., 0.]]),
 array([[1., 4., 6., 4., 4.],
        [2., 0., 7., 9., 1.]])]
```

- Similarly, `vsplit` for vertical split

In [40]:
```
np.vsplit(a,2) # Split a vertically
```

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

In [41]:
```
np.split(a,2) # behaves like vsplit
```

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

Reshaping arrays

- Can change the *shape* of an array if the dimensions match

In [42]:
```
a.shape
```

Out[42]:
```
(2, 12)
```

In [43]:
```
a.shape = 2,12
a
```

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

In [44]:
```
a.shape = 4,6
a
```

Out[44]:
```
array([[1., 1., 6., 1., 7., 2.],
       [8., 1., 4., 6., 4., 4.],
       [0., 9., 3., 7., 6., 7.],
       [0., 2., 0., 7., 9., 1.]])
```

In [45]:
```
a.shape = 3,8
a
```

Out[45]:
```
array([[1., 1., 6., 1., 7., 2., 8., 1.],
       [4., 6., 4., 4., 0., 9., 3., 7.],
       [6., 7., 0., 2., 0., 7., 9., 1.]])
```

In [46]:
```
a.shape = 2,3,4
a
```

Out[46]:
```
array([[[1., 1., 6., 1.],
        [7., 2., 8., 1.],
        [4., 6., 4., 4.]],

       [[0., 9., 3., 7.],
        [6., 7., 0., 2.],
        [0., 7., 9., 1.]]])
```

Copy and view

In [47]:
```
a.shape = 2,12
```

In [48]:
```
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 [49]:
```
a, c, d, e
```

Out[49]:
```
(array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]),
 array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]),
 array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]),
 array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]))
```

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

In [50]:
```
c[0,4] = 88  # Note, not c[0][4]
a, c, d, e
```

Out[50]:
```
(array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]),
 array([[1., 1., 6., 1., 7., 2., 8., 1., 4., 6., 4., 4.],
        [0., 9., 3., 7., 6., 7., 0., 2., 0., 7., 9., 1.]]))
```

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

In [51]:
```
d[0,5] = 66
a, c, d, e
```

Out[51]:
```
(array([[ 1.,  1.,  6.,  1.,  7., 66.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]))
```

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

In [52]:
```
e[0,6] = 77
a, c, d, e
```

Out[52]:
```
(array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]))
```

- We can flatten an array into a sequence

In [53]:
```
for i in a.flat:
    print(i,end=" ")
```

```
1.0 1.0 6.0 1.0 7.0 66.0 77.0 1.0 4.0 6.0 4.0 4.0 0.0 9.0 3.0 7.0 6.0 7.0
0.0 2.0 0.0 7.0 9.0 1.0
```

- `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 [54]:
```
a.base, c.base, d.base, e.base
```

Out[54]:
```
(None,
 None,
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 None)
```

In [55]:
```
d.base is a
```

Out[55]:
```
True
```

- Changing the shape of the base does array not affect the shape of a view

In [56]:
```
a.shape = 4,6
a,c,d,e
```

Out[56]:
```
(array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [ 0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [ 0.,  2.,  0.,  7.,  9.,  1.]]))
```

- Changing the shape of the view does not affect the shape of the base array

In [57]:
```
d.shape = 3,8
a,c,d,e
```

Out[57]:
```
(array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [ 0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.],
        [ 4.,  6.,  4.,  4.,  0.,  9.,  3.,  7.],
        [ 6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [ 0.,  2.,  0.,  7.,  9.,  1.]]))
```

- Since they still have the same base, updating `a` changes the corresponding element in `d`

In [58]:
```
a[3,0] = 99
a,c,d,e
```

Out[58]:
```
(array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [99.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.],
        [ 4.,  6.,  4.,  4.,  0.,  9.,  3.,  7.],
        [ 6.,  7., 99.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66.],
        [77.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.],
        [99.,  2.,  0.,  7.,  9.,  1.]]))
```

- Likewise, updating the shape of `e`, which is an alias for `a`, does not affect the shape of the view `d`

In [59]:
```
e.shape = 8,3
a,c,d,e
```

Out[59]:
```
(array([[ 1.,  1.,  6.],
        [ 1.,  7., 66.],
        [77.,  1.,  4.],
        [ 6.,  4.,  4.],
        [ 0.,  9.,  3.],
        [ 7.,  6.,  7.],
        [99.,  2.,  0.],
        [ 7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1., 88.,  2.,  8.,  1.,  4.,  6.,  4.,  4.],
        [ 0.,  9.,  3.,  7.,  6.,  7.,  0.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.,  1.,  7., 66., 77.,  1.],
        [ 4.,  6.,  4.,  4.,  0.,  9.,  3.,  7.],
        [ 6.,  7., 99.,  2.,  0.,  7.,  9.,  1.]]),
 array([[ 1.,  1.,  6.],
        [ 1.,  7., 66.],
        [77.,  1.,  4.],
        [ 6.,  4.,  4.],
        [ 0.,  9.,  3.],
        [ 7.,  6.,  7.],
        [99.,  2.,  0.],
        [ 7.,  9.,  1.]]))
```

Matrix operations

In [60]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
```

In [61]:
```
a,b
```

Out[61]:
```
(array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]]))
```

- Pointwise addition and multiplication

In [62]:
```
a+b, a*b
```

Out[62]:
```
(array([[ 6,  8],
        [10, 12]]),
 array([[ 5, 12],
        [21, 32]]))
```

- Matrix multiplication

In [63]:
```
np.matmul(a,b)
```

Out[63]:
```
array([[19, 22],
       [43, 50]])
```

- Transpose and inverse

In [64]:
```
a.T
```

Out[64]:
```
array([[1, 3],
       [2, 4]])
```

In [65]:
```
np.linalg.inv(a)
```

Out[65]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

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

In [66]:
```
np.matmul(a,np.linalg.inv(a))
```

Out[66]:
```
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+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 [67]:
```
predictions = np.array([1.2,2.3,3.1]) # (f(x1), f(x2), f(x3))
values = np.array([1,2.5,3]) # (y1, y2, y3)
```

- `predictions - values` creates the array $(f(x_1) - y_1, f(x_2) - y_2, f(x_3) - y_3)$
- Applying `np.square` to this squares each element
- Applying `np.sum` to the result sums up the squares of the errors

In [68]:
```
n = len(predictions)
mse = 1/n * np.sum(np.square(predictions-values))
mse
```

Out[68]:
```
np.float64(0.03000000000000002)
```

Axes

In [69]:
```
a = np.random.random((4,3))*10
a
```

Out[69]:
```
array([[0.74535302, 7.32186747, 3.09968508],
       [1.4351353 , 5.3403798 , 2.05243956],
       [0.17113068, 2.72845255, 0.07238354],
       [2.31854671, 6.79866632, 4.2397082 ]])
```

- `np.sum()` adds up all the entries in the array

In [70]:
```
np.sum(a)
```

Out[70]:
```
np.float64(36.32374822114578)
```

- Selecting `axis = 0` applies the operation column by column

In [71]:
```
np.sum(a,axis=0)
```

Out[71]:
```
array([ 4.67016571, 22.18936614,  9.46421637])
```

- Selecting `axis = 1` applies the opertion row by row

In [72]:
```
np.sum(a,axis=1)
```

Out[72]:
```
array([11.16690556,  8.82795466,  2.97196677, 13.35692123])
```

- `concatenate()` generalizes `vstack()` and `hstack()`
- By default it is `vstack()`, equivalent to setting `axis = 0`
- If we set `axis = 1`, we get `hstack()`

In [73]:
```
b = np.random.random((4,3))*10
b
```

Out[73]:
```
array([[2.78302014, 7.96796426, 5.57485252],
       [9.52754737, 8.00566296, 3.4566572 ],
       [7.23662734, 3.12841927, 1.92923489],
       [8.92895264, 3.02463578, 6.91495572]])
```

In [74]:
```
np.vstack((a,b)), np.hstack((a,b))
```

Out[74]:
```
(array([[0.74535302, 7.32186747, 3.09968508],
        [1.4351353 , 5.3403798 , 2.05243956],
        [0.17113068, 2.72845255, 0.07238354],
        [2.31854671, 6.79866632, 4.2397082 ],
        [2.78302014, 7.96796426, 5.57485252],
        [9.52754737, 8.00566296, 3.4566572 ],
        [7.23662734, 3.12841927, 1.92923489],
        [8.92895264, 3.02463578, 6.91495572]]),
 array([[0.74535302, 7.32186747, 3.09968508, 2.78302014, 7.96796426,
         5.57485252],
        [1.4351353 , 5.3403798 , 2.05243956, 9.52754737, 8.00566296,
         3.4566572 ],
        [0.17113068, 2.72845255, 0.07238354, 7.23662734, 3.12841927,
         1.92923489],
        [2.31854671, 6.79866632, 4.2397082 , 8.92895264, 3.02463578,
         6.91495572]]))
```

In [75]:
```
np.concatenate((a,b))  # Implicitly, axis = 0, so vstack()
```

Out[75]:
```
array([[0.74535302, 7.32186747, 3.09968508],
       [1.4351353 , 5.3403798 , 2.05243956],
       [0.17113068, 2.72845255, 0.07238354],
       [2.31854671, 6.79866632, 4.2397082 ],
       [2.78302014, 7.96796426, 5.57485252],
       [9.52754737, 8.00566296, 3.4566572 ],
       [7.23662734, 3.12841927, 1.92923489],
       [8.92895264, 3.02463578, 6.91495572]])
```

In [76]:
```
np.concatenate((a,b),axis=1) # Concatenate row-wise, so same as hstack()
```

Out[76]:
```
array([[0.74535302, 7.32186747, 3.09968508, 2.78302014, 7.96796426,
        5.57485252],
       [1.4351353 , 5.3403798 , 2.05243956, 9.52754737, 8.00566296,
        3.4566572 ],
       [0.17113068, 2.72845255, 0.07238354, 7.23662734, 3.12841927,
        1.92923489],
       [2.31854671, 6.79866632, 4.2397082 , 8.92895264, 3.02463578,
        6.91495572]])
```

In [77]:
```
c = np.random.random((1,7))
d = np.random.random((1,7))
c,d
```

Out[77]:
```
(array([[0.77252961, 0.9476818 , 0.22204027, 0.11708873, 0.91664732,
         0.35529799, 0.09111515]]),
 array([[0.3056627 , 0.83884962, 0.25602509, 0.06546634, 0.41950657,
         0.73748701, 0.01506722]]))
```

In [78]:
```
np.concatenate((c,d),axis=1)
```

Out[78]:
```
array([[0.77252961, 0.9476818 , 0.22204027, 0.11708873, 0.91664732,
        0.35529799, 0.09111515, 0.3056627 , 0.83884962, 0.25602509,
        0.06546634, 0.41950657, 0.73748701, 0.01506722]])
```

Broadcasting

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

In [79]:
```
a = np.array([1.0, 2.0, 3.0])
b = 2.0
a * b
```

Out[79]:
```
array([2., 4., 6.])
```

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

In [80]:
```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
a * b
```

Out[80]:
```
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 [81]:
```
pic = np.random.random((4,4,3))*10
```

In [82]:
```
pic
```

Out[82]:
```
array([[[5.7448521 , 6.21715157, 8.09743372],
        [3.86616619, 1.83400169, 3.30234273],
        [7.14510501, 8.53592886, 0.1984932 ],
        [2.83792466, 2.78075699, 6.47495444]],

       [[4.20196209, 0.05568092, 9.82591891],
        [7.88589042, 4.24747824, 1.91112919],
        [6.23340351, 5.32555818, 8.30908835],
        [0.10657329, 6.2069235 , 3.47629843]],

       [[4.79882427, 0.91260498, 4.87367519],
        [5.43909663, 9.33695084, 8.74619775],
        [7.92382402, 0.32737941, 0.68173562],
        [1.91816694, 3.49747929, 7.85137433]],

       [[8.97653917, 4.78902054, 5.36404235],
        [5.54700431, 9.70798109, 8.02925934],
        [8.83133383, 0.98081094, 4.92779854],
        [3.86833919, 4.14670514, 6.73500368]]])
```

In [83]:
```
pic*[1,100,1000]
```

Out[83]:
```
array([[[5.74485210e+00, 6.21715157e+02, 8.09743372e+03],
        [3.86616619e+00, 1.83400169e+02, 3.30234273e+03],
        [7.14510501e+00, 8.53592886e+02, 1.98493199e+02],
        [2.83792466e+00, 2.78075699e+02, 6.47495444e+03]],

       [[4.20196209e+00, 5.56809248e+00, 9.82591891e+03],
        [7.88589042e+00, 4.24747824e+02, 1.91112919e+03],
        [6.23340351e+00, 5.32555818e+02, 8.30908835e+03],
        [1.06573294e-01, 6.20692350e+02, 3.47629843e+03]],

       [[4.79882427e+00, 9.12604978e+01, 4.87367519e+03],
        [5.43909663e+00, 9.33695084e+02, 8.74619775e+03],
        [7.92382402e+00, 3.27379407e+01, 6.81735625e+02],
        [1.91816694e+00, 3.49747929e+02, 7.85137433e+03]],

       [[8.97653917e+00, 4.78902054e+02, 5.36404235e+03],
        [5.54700431e+00, 9.70798109e+02, 8.02925934e+03],
        [8.83133383e+00, 9.80810942e+01, 4.92779854e+03],
        [3.86833919e+00, 4.14670514e+02, 6.73500368e+03]]])
```

Note

- In the example above, a shape of `(3,4,4)` would display more intuitively as 3 layers of colour for the base image of shape `(4,4)`

- However, for broadcasting, we need the *rightmost* index to match, so we wrote it as `(4,4,3)`, which is laid out less intuitively by `numpy` as 4 layers with shape `(4,3)`

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

- Here is the whole computation in one shot

In [84]:
```
observation = np.array([111.0, 188.0])  # (x,y)
codes = np.array([[132.0, 193.0],  # [(x1,y1), (x2,y2), (x3,y3), (x4,y4)]
                  [102.0, 203.0],
                  [45.0, 155.0],
                  [57.0, 173.0]])

diff = codes - observation
dist = np.sqrt(np.sum(diff**2,axis=1))
dist, np.argmin(dist)
```

Out[84]:
```
(array([21.58703314, 17.49285568, 73.79024326, 56.04462508]), np.int64
(1))
```

- Let us break this up into steps

1. The setup

In [85]:
```
observation = np.array([111.0, 188.0])  # (x,y)
codes = np.array([[132.0, 193.0],  # [(x1,y1), (x2,y2), (x3,y3), (x4,y4)]
                  [102.0, 203.0],
                  [45.0, 155.0],
                  [57.0, 173.0]])
```

2. Use broadcast to subtact (x, y) from each (x_i, y_i) to get the array $[(x_1 - x, y_1 - y), (x_2 - x, y_2 - y), (x_3 - x, y_3 - y), (x_4 - x, y_4 - y)]$

In [86]:
```
diff = codes - observation
```

In [87]:
```
diff
```

Out[87]:
```
array([[ 21.,   5.],
       [ -9.,  15.],
       [-66., -33.],
       [-54., -15.]])
```

3. Use scalar broadcast to map each pair $(x_i - x, y_i - y)$ to $((x_i - x)^2, (y_i - y)^2)$

In [88]:
```
diff**2
```

Out[88]:
```
array([[ 441.,   25.],
       [  81.,  225.],
       [4356., 1089.],
       [2916.,  225.]])
```

4. Add up each row as $(x_i - x)^2 + (y_i - y)^2$ by computing `np.sum()` along `axis = 1`

In [89]:
```
np.sum(diff**2,axis=1)
```

Out[89]:
```
array([ 466.,  306., 5445., 3141.])
```

5. Apply `np.sqrt()` pointwise to this array of squared differences

In [90]:
```
np.sqrt(np.sum(diff**2,axis=1))
```

Out[90]:
```
array([21.58703314, 17.49285568, 73.79024326, 56.04462508])
```

6. Find the index where this value is minimum

In [91]:
```
dist = np.sqrt(np.sum(diff**2,axis=1))
np.argmin(dist)
```

Out[91]:
```
np.int64(1)
```

- Or, equivalently, without the intermediate variable `dist`

In [92]:
```
np.argmin(np.sqrt(np.sum(diff**2,axis=1)))
```

Out[92]:
```
np.int64(1)
```

- In fact, the entire computation can be written in one line

In [93]:
```
np.argmin(np.sqrt(np.sum((codes-observation)**2,axis=1)))
```

Out[93]:
```
np.int64(1)
```

In [94]:
```
np.min(np.sqrt(np.sum((codes-observation)**2,axis=1)))
```

Out[94]:
```
np.float64(17.4928556845359)
```