

Lecture 17, 15 October 2026

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 must 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]],[[9,5,4],[6,7,5]]])
      d
```

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

- An element of a multidimensional array is `a[i,j,k]` not `a[i][j][k]`

```
In [8]: d[1,0,2]
```

```
Out[8]: np.int64(4)
```

- Create a zero array

```
In [9]: m_zeros = np.zeros((5,3)) # np.zeros(shape), shape = (dim1,dim2,...,dimk)
      m_zeros
```

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

```
In [10]: m_zeros.dtype
```

```
Out[10]: dtype('float64')
```

```
In [11]: m_zeros = np.zeros(5,3) # The shape must be a sequence, the argument
```

```

-----
-
TypeError                                Traceback (most recent call last)
Cell In[11], line 1
----> 1 m_zeros = np.zeros(5,3)    # The shape must be a sequence, the argument is a single sequence

TypeError: Cannot interpret '3' as a data type

```

```

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

```

```

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

```

- An array of 1's

```

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.]])

```

- Arrays with random numbers

```

In [14]: m_rand = np.random.random((4,6)) # Implicitly uniform in [0,1)
         m_rand

```

```

Out[14]: array([[0.00287431, 0.29674119, 0.24476656, 0.18877922, 0.99492982,
                  0.44853876],
                 [0.6345094 , 0.47673303, 0.4519863 , 0.83850117, 0.86138401,
                  0.70540875],
                 [0.79674745, 0.46600115, 0.72986825, 0.83545426, 0.78176263,
                  0.14423092],
                 [0.89425469, 0.65231874, 0.5992051 , 0.34482752, 0.83451738,
                  0.04742279]])

```

```
In [15]: m_rand_int = np.random.randint(1,100,(3,4)) # Integers uniformly from [0, m_rand_int
```

```
Out[15]: array([[34, 72, 62, 40],
               [67,  3, 72, 58],
               [87,  7, 82, 79]])
```

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

```
Out[16]: array([[ -0.93927631,  0.43362544],
               [ -0.85580182, -0.22417338],
               [ 0.6479333 ,  0.18524646],
               [-0.63437082,  1.18033941],
               [-0.75870278, -0.24999268]])
```

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 $m \times n$ 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. 6.]
 [7. 7.]]
[[4. 0.]
 [0. 2.]
 [5. 8.]]
```

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

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

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

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

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

```
Out[26]: array([[7., 2., 5.],
               [0., 5., 6.]])
```

```
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:
290, in vstack(tup, dtype, casting)
    288 if not isinstance(arrs, tuple):
    289     arrs = (arrs,)
--> 290 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([[0., 7.],  
               [4., 6.],  
               [5., 4.]])
```

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

```
Out[29]: array([[4., 6.],  
               [7., 7.],  
               [0., 7.],  
               [4., 6.],  
               [5., 4.]])
```

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

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

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

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

```
Out[31]: array([[4., 6.],  
               [7., 7.],  
               [4., 0.],  
               [0., 2.],  
               [5., 8.],  
               [0., 7.],  
               [4., 6.],  
               [5., 4.]])
```

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

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

```

-----
ValueError                                Traceback (most recent call last)
Cell In[32], line 1
----> 1 np.hstack((a,b))

File ~/python-venv/lib/python3.13/site-packages/numpy/_core/shape_base.py:
365, in hstack(tup, dtype, casting)
    363     return _nx.concatenate(arrs, 0, dtype=dtype, casting=casting)
    364 else:
--> 365     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 [33]: np.hstack((b,d))
```

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

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

```
In [34]: e = np.floor(10*np.random.random((3,3)))
e
```

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

```
In [35]: np.hstack((b,d,e))
```

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

Splitting arrays

```
In [36]: a = np.floor(10*np.random.random((2,12)))
a
```

```
Out[36]: array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],
               [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.]])
```

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

```
In [37]: np.hsplit(a,3)
```

```
Out[37]: [array([[4., 7., 4., 7.],
                [4., 8., 6., 9.]]) ,
          array([[6., 2., 9., 9.],
                [7., 6., 8., 9.]]) ,
          array([[0., 5., 0., 9.],
                [6., 3., 6., 7.]])]
```

```
In [38]: np.hsplit(a,6)
```

```
Out[38]: [array([[4., 7.],
                [4., 8.]]) ,
          array([[4., 7.],
                [6., 9.]]) ,
          array([[6., 2.],
                [7., 6.]]) ,
          array([[9., 9.],
                [8., 9.]]) ,
          array([[0., 5.],
                [6., 3.]]) ,
          array([[0., 9.],
                [6., 7.]])]
```

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

```
-----
ValueError                                Traceback (most recent call last)
Cell In[39], line 1
----> 1 np.hsplit(a,5)

File ~/python-venv/lib/python3.13/site-packages/numpy/lib/_shape_base_impl.py:929, in hsplit(ary, indices_or_sections)
    927     raise ValueError('hsplit only works on arrays of 1 or more dimensions')
    928 if ary.ndim > 1:
--> 929     return split(ary, indices_or_sections, 1)
    930 else:
    931     return split(ary, indices_or_sections, 0)

File ~/python-venv/lib/python3.13/site-packages/numpy/lib/_shape_base_impl.py:854, in split(ary, indices_or_sections, axis)
    852     N = ary.shape[axis]
    853     if N % sections:
--> 854         raise ValueError(
    855             'array split does not result in an equal division') from None
    856 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 [40]: np.hsplit(a,(2,5,7)) # a[:2], a[2:5], a[5:7], a[7:]
```

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

- Similarly, `vsplit` for vertical split

```
In [41]: np.vsplit(a,2) # Split a vertically
```

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

```
In [42]: np.split(a,2) # behaves like vsplit
```

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

Reshaping arrays

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

```
In [43]: a.shape
```

```
Out[43]: (2, 12)
```

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

```
/tmp/ipykernel_46357/1043616937.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.
As an alternative, you can create a new view using np.reshape (with copy=False if needed).
a.shape = 4,6
```

```
Out[44]: array([[4., 7., 4., 7., 6., 2.],
                [9., 9., 0., 5., 0., 9.],
                [4., 8., 6., 9., 7., 6.],
                [8., 9., 6., 3., 6., 7.]])
```

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

```
/tmp/ipykernel_46357/2042295483.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.
As an alternative, you can create a new view using np.reshape (with copy=False if needed).
a.shape = 3,8
```

```
Out[45]: array([[4., 7., 4., 7., 6., 2., 9., 9.],
                [0., 5., 0., 9., 4., 8., 6., 9.],
                [7., 6., 8., 9., 6., 3., 6., 7.]])
```

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

```
a
```

```
/tmp/ipykernel_46357/4131849822.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.  
As an alternative, you can create a new view using np.reshape (with copy=False if needed).  
a.shape = 2,3,4
```

```
Out[46]: array([[4., 7., 4., 7.],  
               [6., 2., 9., 9.],  
               [0., 5., 0., 9.]],  
  
          [[4., 8., 6., 9.],  
           [7., 6., 8., 9.],  
           [6., 3., 6., 7.]])
```

Copy and view

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

```
/tmp/ipykernel_46357/3929540922.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.  
As an alternative, you can create a new view using np.reshape (with copy=False if needed).  
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([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
                [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.])),  
 array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
        [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.])),  
 array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
        [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.])),  
 array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
        [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.]])
```

- 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([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
                [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.])),  
 array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],  
        [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.])),  
 array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
        [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.])),  
 array([[4., 7., 4., 7., 6., 2., 9., 9., 0., 5., 0., 9.],  
        [4., 8., 6., 9., 7., 6., 8., 9., 6., 3., 6., 7.]])
```

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

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

```
Out[51]: (array([[ 4.,  7.,  4.,  7.,  6., 66.,  9.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7.,  6., 66.,  9.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7.,  6., 66.,  9.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.])))
```

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

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

```
Out[52]: (array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.])))
```

- We can flatten an array into a sequence

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

```
4.0 7.0 4.0 7.0 6.0 66.0 77.0 9.0 0.0 5.0 0.0 9.0 4.0 8.0 6.0 9.0 7.0 6.0
8.0 9.0 6.0 3.0 6.0 7.0
```

- `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([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.,  0.,  5.,  0.,  9.],
                [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          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
```

```
/tmp/ipykernel_46357/1774912613.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.  
As an alternative, you can create a new view using np.reshape (with copy=False if needed).  
a.shape = 4,6
```

```
Out[56]: (array([[ 4.,  7.,  4.,  7.,  6., 66.],  
               [77.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.],  
               [ 8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7.,  6., 66.],  
               [77.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.],  
               [ 8.,  9.,  6.,  3.,  6.,  7.]])
```

- 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
```

```
/tmp/ipykernel_46357/2378460131.py:1: DeprecationWarning: Setting the shape on a NumPy array has been deprecated in NumPy 2.5.  
As an alternative, you can create a new view using np.reshape (with copy=False if needed).  
d.shape = 3,8
```

```
Out[57]: (array([[ 4.,  7.,  4.,  7.,  6., 66.],  
               [77.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.],  
               [ 8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.],  
               [ 0.,  5.,  0.,  9.,  4.,  8.,  6.,  9.],  
               [ 7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),  
         array([[ 4.,  7.,  4.,  7.,  6., 66.],  
               [77.,  9.,  0.,  5.,  0.,  9.],  
               [ 4.,  8.,  6.,  9.,  7.,  6.],  
               [ 8.,  9.,  6.,  3.,  6.,  7.]])
```

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

- 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
```

```
/tmp/ipykernel_46357/682286037.py:1: DeprecationWarning: Setting the shape
on a NumPy array has been deprecated in NumPy 2.5.
As an alternative, you can create a new view using np.reshape (with copy=F
also if needed).
e.shape = 8,3
```

```
Out[59]: (array([[ 4.,  7.,  4.],
                  [ 7.,  6., 66.],
                  [77.,  9.,  0.],
                  [ 5.,  0.,  9.],
                  [ 4.,  8.,  6.],
                  [ 9.,  7.,  6.],
                  [99.,  9.,  6.],
                  [ 3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7., 88.,  2.,  9.,  9.,  0.,  5.,  0.,  9.],
                  [ 4.,  8.,  6.,  9.,  7.,  6.,  8.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.,  7.,  6., 66., 77.,  9.],
                  [ 0.,  5.,  0.,  9.,  4.,  8.,  6.,  9.],
                  [ 7.,  6., 99.,  9.,  6.,  3.,  6.,  7.]]),
          array([[ 4.,  7.,  4.],
                  [ 7.,  6., 66.],
                  [77.,  9.,  0.],
                  [ 5.,  0.,  9.],
                  [ 4.,  8.,  6.],
                  [ 9.,  7.,  6.],
                  [99.,  9.,  6.],
                  [ 3.,  6.,  7.]])
```

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([[1.00000000e+00,  0.00000000e+00],
                [8.8817842e-16,  1.00000000e+00]])
```

- Fit a function f to a set of data points $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$
- 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(0.030000000000000002)
```

Axes

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

```
Out[69]: array([[0.81918247, 0.66135039, 5.62839188],
               [1.01971084, 7.75736105, 8.94883196],
               [8.05270082, 5.09213257, 0.0812723 ],
               [8.45844436, 5.28911391, 1.0152308 ]])
```

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

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

```
Out[70]: np.float64(52.823723341821086)
```

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

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

```
Out[71]: array([18.3500385 , 18.7999579 , 15.67372694])
```

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

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

```
Out[72]: array([ 7.10892474, 17.72590385, 13.22610568, 14.76278907])
```

- `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([[0.76220839, 5.13172146, 8.54228238],
               [8.78106453, 0.50656321, 6.77683995],
               [7.489932 , 9.45161404, 0.88572984],
               [7.89212861, 2.33473777, 1.15055342]])
```

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

```
Out[74]: (array([[0.81918247, 0.66135039, 5.62839188],
                [1.01971084, 7.75736105, 8.94883196],
                [8.05270082, 5.09213257, 0.0812723 ],
                [8.45844436, 5.28911391, 1.0152308 ],
                [0.76220839, 5.13172146, 8.54228238],
                [8.78106453, 0.50656321, 6.77683995],
                [7.489932 , 9.45161404, 0.88572984],
                [7.89212861, 2.33473777, 1.15055342]]),
         array([[0.81918247, 0.66135039, 5.62839188, 0.76220839, 5.13172146,
                8.54228238],
                [1.01971084, 7.75736105, 8.94883196, 8.78106453, 0.50656321,
                6.77683995],
                [8.05270082, 5.09213257, 0.0812723 , 7.489932 , 9.45161404,
                0.88572984],
                [8.45844436, 5.28911391, 1.0152308 , 7.89212861, 2.33473777,
                1.15055342]]))
```

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

```
Out[75]: array([[0.81918247, 0.66135039, 5.62839188],
                [1.01971084, 7.75736105, 8.94883196],
                [8.05270082, 5.09213257, 0.0812723 ],
                [8.45844436, 5.28911391, 1.0152308 ],
                [0.76220839, 5.13172146, 8.54228238],
                [8.78106453, 0.50656321, 6.77683995],
                [7.489932 , 9.45161404, 0.88572984],
                [7.89212861, 2.33473777, 1.15055342]])
```

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

```
Out[76]: array([[0.81918247, 0.66135039, 5.62839188, 0.76220839, 5.13172146,
                8.54228238],
                [1.01971084, 7.75736105, 8.94883196, 8.78106453, 0.50656321,
                6.77683995],
                [8.05270082, 5.09213257, 0.0812723 , 7.489932 , 9.45161404,
                0.88572984],
                [8.45844436, 5.28911391, 1.0152308 , 7.89212861, 2.33473777,
                1.15055342]])
```

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

```
Out[77]: (array([[0.88046795, 0.99392296, 0.27633746, 0.73703386, 0.29114019,
                0.66363698, 0.67184984]]),
         array([[0.00581885, 0.83850564, 0.20357386, 0.05392887, 0.88108961,
                0.41618057, 0.26064871]]))
```

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

```
Out[78]: array([[0.88046795, 0.99392296, 0.27633746, 0.73703386, 0.29114019,
                0.66363698, 0.67184984, 0.00581885, 0.83850564, 0.20357386,
                0.05392887, 0.88108961, 0.41618057, 0.26064871]])
```

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([[9.54339525, 2.04858848, 8.32272994],
                [5.80180354, 9.62061036, 7.18816488],
                [1.37215031, 9.76922239, 3.1358162 ],
                [5.50714082, 5.64770232, 3.69201246]],

                [[9.06901035, 7.25147347, 6.80622958],
                [4.96124661, 5.98674093, 6.85115847],
                [1.96223355, 6.27720441, 4.39818831],
                [3.47650489, 9.69669405, 8.54726802]],

                [[6.13447243, 7.4298857 , 0.18210238],
                [7.28022702, 8.3608639 , 0.55779384],
                [3.22308057, 2.4264986 , 0.51528485],
                [9.20997866, 3.60708512, 5.33017625]],

                [[7.19704387, 8.7503995 , 2.7930906 ],
                [2.98163554, 2.92389833, 7.62467763],
                [1.3500405 , 9.1557671 , 0.96891741],
                [1.14362885, 4.76294657, 9.21407576]]])
```

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

```
Out[83]: array([[9.54339525e+00, 2.04858848e+02, 8.32272994e+03],
               [5.80180354e+00, 9.62061036e+02, 7.18816488e+03],
               [1.37215031e+00, 9.76922239e+02, 3.13581620e+03],
               [5.50714082e+00, 5.64770232e+02, 3.69201246e+03]],

               [[9.06901035e+00, 7.25147347e+02, 6.80622958e+03],
               [4.96124661e+00, 5.98674093e+02, 6.85115847e+03],
               [1.96223355e+00, 6.27720441e+02, 4.39818831e+03],
               [3.47650489e+00, 9.69669405e+02, 8.54726802e+03]],

               [[6.13447243e+00, 7.42988570e+02, 1.82102382e+02],
               [7.28022702e+00, 8.36086390e+02, 5.57793844e+02],
               [3.22308057e+00, 2.42649860e+02, 5.15284854e+02],
               [9.20997866e+00, 3.60708512e+02, 5.33017625e+03]],

               [[7.19704387e+00, 8.75039950e+02, 2.79309060e+03],
               [2.98163554e+00, 2.92389833e+02, 7.62467763e+03],
               [1.35004050e+00, 9.15576710e+02, 9.68917413e+02],
               [1.14362885e+00, 4.76294657e+02, 9.21407576e+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 $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$, 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
- Let us break this up into steps

1. The setup

```
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]])
```

2. Use broadcast to subtract (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 [85]: diff = codes - observation
```

```
In [86]: diff
```

```
Out[86]: 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 [87]: diff**2
```

```
Out[87]: 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 [88]: np.sum(diff**2,axis=1)
```

```
Out[88]: array([ 466.,  306., 5445., 3141.] )
```

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

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

```
Out[89]: array([21.58703314, 17.49285568, 73.79024326, 56.04462508])
```

6. Find the index where this value is minimum

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

```
Out[90]: np.int64(1)
```

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

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

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

- Here is the whole computation in one shot

```
In [92]: 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.argmax(dist)
```

```
Out[92]: (array([21.58703314, 17.49285568, 73.79024326, 56.04462508]), np.int64(1))
```

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

```
In [93]: np.argmax(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)
```