Lecture 13, 20 September 2022

Using numpy

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

Load numpy

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

Create an array from a sequence

- Note that a string is considered a single value, not a sequence

In [2]:
```
a = np.array([1,2,3])
b = np.array(range(10))
c = np.array(["strong"])
d = np.array(("a","b","c"))
a, b, c, d
```

Out[2]:
```
(array([1, 2, 3]),
 array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),
 array(['strong'], dtype='<U6'),
 array(['a', 'b', 'c'], dtype='<U1'))
```

- `array.dtype` gives the datatype of the underlying array

In [3]:
```
a.dtype, b.dtype, c.dtype, d.dtype
```

Out[3]:
```
(dtype('int64'), dtype('int64'), dtype('<U6'), dtype('<U1'))
```

- For strings, `<UN` indicates a Unicode string of length `N`
- Assigning a longer value will truncate to the given length

In [4]:
```
c[0] ="verylongstring"
c
```

Out[4]:
```
array(['verylo'], dtype='<U6')
```

- It is possible to have a numpy array with different size members by specifying the type to be `object`
- It is also possible to change the type of an array
- See the numpy documentation

Indexing and slicing

In [5]:
```
a = np.arange(10)
```

In [6]:
```
a[2], a[2:5]
```

Out[6]:
```
(2, array([2, 3, 4]))
```

In [7]:
```
a[:6:2] = -1000  # equivalent to a[0:6:2] = -1000
a
```

Out[7]:
```
array([-1000,     1, -1000,     3, -1000,     5,     6,     7,     8,
           9])
```

In [8]:
```
def f(x,y):
    return(10*x +  y)
b = np.fromfunction(f,(5,4))
b
```

Out[8]:
```
array([[ 0.,  1.,  2.,  3.],
       [10., 11., 12., 13.],
       [20., 21., 22., 23.],
       [30., 31., 32., 33.],
       [40., 41., 42., 43.]])
```

- Default `dtype` is `float16``
- Can provided preferred `dtype` when constructing the array

In [9]:
```
b.dtype
```

Out[9]:
```
dtype('float64')
```

In [10]:
```
def f(x,y):
    return(10*x +  y)
b = np.fromfunction(f,(5,4),dtype=int)
b, b.dtype
```

Out[10]:
```
(array([[ 0,  1,  2,  3],
        [10, 11, 12, 13],
        [20, 21, 22, 23],
        [30, 31, 32, 33],
        [40, 41, 42, 43]]),
 dtype('int64'))
```

In [11]:
```
b[2,3]  # Not b[2][3]
```

Out[11]:
```
23
```

In [12]:
```
b[0:5, 1] # second column from each row of b
```

Out[12]:
```
array([ 1, 11, 21, 31, 41])
```

In [13]:
```
b[ : ,1]  # equivalent to the previous example
```

Out[13]:
```
array([ 1, 11, 21, 31, 41])
```

In [14]:
```
b[1:3, :]  # each column in the second and third row of b
```

Out[14]:
```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

In [15]:
```
b[1:4,1:3]
```

Out[15]:
```
array([[11, 12],
       [21, 22],
       [31, 32]])
```

Iterating over elements

In [16]:
```
print(b)
```

```
[[ 0  1  2  3]
 [10 11 12 13]
 [20 21 22 23]
 [30 31 32 33]
 [40 41 42 43]]
```

In [17]:
```
for row in b:
    print(row)
```

```
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

In [18]:
```
for element in b.flat:
    print(element,end=' ')
```

```
0 1 2 3 10 11 12 13 20 21 22 23 30 31 32 33 40 41 42 43
```

In [19]:
```
list(b.flat)
```

Out[19]:
```
[0, 1, 2, 3, 10, 11, 12, 13, 20, 21, 22, 23, 30, 31, 32, 33, 40, 41, 42, 43]
```

- Technically `b.flat` produces an *iterator* that allows you to iterate through the array
- Each call to `b.flat` generates an independent iterator, so that you can execute a nested loop as you would expect

In [20]:
```
for x in b.flat:
    for y in b.flat:
        print(x,y, end=", ")
    print()
```

```
0 0, 0 1, 0 2, 0 3, 0 10, 0 11, 0 12, 0 13, 0 20, 0 21, 0 22, 0 23, 0 30, 0 31, 0 32, 0 33, 0 40, 0 41, 0 42, 0 43,
1 0, 1 1, 1 2, 1 3, 1 10, 1 11, 1 12, 1 13, 1 20, 1 21, 1 22, 1 23, 1 30, 1 31, 1 32, 1 33, 1 40, 1 41, 1 42, 1 43,
2 0, 2 1, 2 2, 2 3, 2 10, 2 11, 2 12, 2 13, 2 20, 2 21, 2 22, 2 23, 2 30, 2 31, 2 32, 2 33, 2 40, 2 41, 2 42, 2 43,
3 0, 3 1, 3 2, 3 3, 3 10, 3 11, 3 12, 3 13, 3 20, 3 21, 3 22, 3 23, 3 30, 3 31, 3 32, 3 33, 3 40, 3 41, 3 42, 3 43,
10 0, 10 1, 10 2, 10 3, 10 10, 10 11, 10 12, 10 13, 10 20, 10 21, 10 22, 10 23, 10 30, 10 31, 10 32, 10 33, 10 40, 10 41, 10
42, 10 43,
11 0, 11 1, 11 2, 11 3, 11 10, 11 11, 11 12, 11 13, 11 20, 11 21, 11 22, 11 23, 11 30, 11 31, 11 32, 11 33, 11 40, 11 41, 11
42, 11 43,
12 0, 12 1, 12 2, 12 3, 12 10, 12 11, 12 12, 12 13, 12 20, 12 21, 12 22, 12 23, 12 30, 12 31, 12 32, 12 33, 12 40, 12 41, 12
42, 12 43,
13 0, 13 1, 13 2, 13 3, 13 10, 13 11, 13 12, 13 13, 13 20, 13 21, 13 22, 13 23, 13 30, 13 31, 13 32, 13 33, 13 40, 13 41, 13
42, 13 43,
20 0, 20 1, 20 2, 20 3, 20 10, 20 11, 20 12, 20 13, 20 20, 20 21, 20 22, 20 23, 20 30, 20 31, 20 32, 20 33, 20 40, 20 41, 20
42, 20 43,
21 0, 21 1, 21 2, 21 3, 21 10, 21 11, 21 12, 21 13, 21 20, 21 21, 21 22, 21 23, 21 30, 21 31, 21 32, 21 33, 21 40, 21 41, 21
42, 21 43,
22 0, 22 1, 22 2, 22 3, 22 10, 22 11, 22 12, 22 13, 22 20, 22 21, 22 22, 22 23, 22 30, 22 31, 22 32, 22 33, 22 40, 22 41, 22
42, 22 43,
23 0, 23 1, 23 2, 23 3, 23 10, 23 11, 23 12, 23 13, 23 20, 23 21, 23 22, 23 23, 23 30, 23 31, 23 32, 23 33, 23 40, 23 41, 23
42, 23 43,
30 0, 30 1, 30 2, 30 3, 30 10, 30 11, 30 12, 30 13, 30 20, 30 21, 30 22, 30 23, 30 30, 30 31, 30 32, 30 33, 30 40, 30 41, 30
42, 30 43,
31 0, 31 1, 31 2, 31 3, 31 10, 31 11, 31 12, 31 13, 31 20, 31 21, 31 22, 31 23, 31 30, 31 31, 31 32, 31 33, 31 40, 31 41, 31
42, 31 43,
32 0, 32 1, 32 2, 32 3, 32 10, 32 11, 32 12, 32 13, 32 20, 32 21, 32 22, 32 23, 32 30, 32 31, 32 32, 32 33, 32 40, 32 41, 32
42, 32 43,
33 0, 33 1, 33 2, 33 3, 33 10, 33 11, 33 12, 33 13, 33 20, 33 21, 33 22, 33 23, 33 30, 33 31, 33 32, 33 33, 33 40, 33 41, 33
42, 33 43,
40 0, 40 1, 40 2, 40 3, 40 10, 40 11, 40 12, 40 13, 40 20, 40 21, 40 22, 40 23, 40 30, 40 31, 40 32, 40 33, 40 40, 40 41, 40
42, 40 43,
41 0, 41 1, 41 2, 41 3, 41 10, 41 11, 41 12, 41 13, 41 20, 41 21, 41 22, 41 23, 41 30, 41 31, 41 32, 41 33, 41 40, 41 41, 41
42, 41 43,
42 0, 42 1, 42 2, 42 3, 42 10, 42 11, 42 12, 42 13, 42 20, 42 21, 42 22, 42 23, 42 30, 42 31, 42 32, 42 33, 42 40, 42 41, 42
42, 42 43,
43 0, 43 1, 43 2, 43 3, 43 10, 43 11, 43 12, 43 13, 43 20, 43 21, 43 22, 43 23, 43 30, 43 31, 43 32, 43 33, 43 40, 43 41, 43
42, 43 43,
```

Stacking arrays

- Create an array of zeroes of type `int`

In [21]:
```
a = np.zeros((5,7),dtype=int)
a
```

Out[21]:
```
array([[0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0, 0, 0]])
```

- Create arrays with random values
- `np.random.random` generates a random number uniformly in $[0, 1)$

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

Out[22]:
```
(array([[6., 6.],
        [0., 9.]]),
 array([[3., 9.],
        [1., 3.]]))
```

- Stack horizontally or vertically, dimensions much match

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

Out[23]:
```
array([[6., 6.],
       [0., 9.],
       [3., 9.],
       [1., 3.],
       [6., 6.],
       [0., 9.]])
```

In [24]:
```
np.hstack((a,b))
```

Out[24]:
```
array([[6., 6., 3., 9.],
       [0., 9., 1., 3.]])
```

Splitting arrays

In [25]:
```
a = np.floor(10*np.random.random((2,12)))
a
```

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

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

In [26]:
```
np.hsplit(a,3)
```

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

In [27]:
```
np.hsplit(a,6)
```

Out[27]:
```
[array([[6., 9.],
        [9., 8.]]),
 array([[9., 5.],
        [8., 3.]]),
 array([[5., 5.],
        [7., 6.]]),
 array([[3., 1.],
        [4., 6.]]),
 array([[4., 3.],
        [8., 3.]]),
 array([[9., 1.],
        [4., 4.]])]
```

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

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/tmp/ipykernel_153781/2675804604.py in <module>
----> 1 np.hsplit(a,5)

<__array_function__ internals> in hsplit(*args, **kwargs)

~/miniconda3/lib/python3.7/site-packages/numpy/lib/shape_base.py in hsplit(ary, indices_or_sections)
    938         raise ValueError('hsplit only works on arrays of 1 or more dimensions')
    939     if ary.ndim > 1:
--> 940         return split(ary, indices_or_sections, 1)
    941     else:
    942         return split(ary, indices_or_sections, 0)

<__array_function__ internals> in split(*args, **kwargs)

~/miniconda3/lib/python3.7/site-packages/numpy/lib/shape_base.py in split(ary, indices_or_sections, axis)
    871         if N % sections:
    872             raise ValueError(
--> 873                 'array split does not result in an equal division') from None
    874     return array_split(ary, indices_or_sections, axis)
    875

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 [29]:
```
np.hsplit(a,(2,5,7)) # a[:2], a[2:5], a[5:7], a[7:]
```

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

- Similarly, `vsplit` for vertical split

In [30]:
```
np.vsplit(a,2) # Split a vertically
```

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

In [31]:
```
np.split(a,2) # behaves like vsplit
```

Out[31]:
```
[array([[6., 9., 9., 5., 5., 5., 3., 1., 4., 3., 9., 1.]]),
 array([[9., 8., 8., 3., 7., 6., 4., 6., 8., 3., 4., 4.]])]
```

Copy and view

In [32]:
```
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 [33]:
```
a, c, d, e
```

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

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

In [34]:
```
c[0,4] = 88
a, c, d, e
```

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

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

In [35]:
```
d[0,5] = 66
a, c, d, e
```

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

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

In [36]:
```
e[0,6] = 77
a, c, d, e
```

Out[36]:
```
(array([[ 6.,  9.,  9.,  5.,  5., 66., 77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5., 88.,  5.,  3.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5.,  5., 66., 77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5.,  5., 66., 77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]))
```

- `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([[ 6.,  9.,  9.,  5.,  5., 66., 77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 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([[ 6.,  9.,  9.,  5.,  5., 66.],
        [77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.],
        [ 4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5., 88.,  5.,  3.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5.,  5., 66., 77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.,  4.,  6.,  8.,  3.,  4.,  4.]]),
 array([[ 6.,  9.,  9.,  5.,  5., 66.],
        [77.,  1.,  4.,  3.,  9.,  1.],
        [ 9.,  8.,  8.,  3.,  7.,  6.],
        [ 4.,  6.,  8.,  3.,  4.,  4.]]))
```

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

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

Matrix operations

In [42]:
```
a = np.array([[1,2],[3,4]])
b = np.array([[5,6],[7,8]])
```

In [43]:
```
a,b
```

Out[43]:
```
(array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]]))
```

- Pointwise addition and multiplication

In [44]:
```
a+b, a*b
```

Out[44]:
```
(array([[ 6,  8],
        [10, 12]]),
 array([[ 5, 12],
        [21, 32]]))
```

- Matrix multiplication

In [45]:
```
np.matmul(a,b)
```

Out[45]:
```
array([[19, 22],
       [43, 50]])
```

- Transpose and inverse

In [46]:
```
a.T
```

Out[46]:
```
array([[1, 3],
       [2, 4]])
```

In [47]:
```
np.linalg.inv(a)
```

Out[47]:
```
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

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

In [48]:
```
np.matmul(a,np.linalg.inv(a))
```

Out[48]:
```
array([[1.00000000e+00, 1.11022302e-16],
       [0.00000000e+00, 1.00000000e+00]])
```