Lecture 12, 21 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]:
```
a = np.array([1,2,3])
a, print(a)
```

```
[1 2 3]
```

Out[2]:
```
(array([1, 2, 3]), None)
```

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

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

In [4]:
```
b = np.array(range(10))
b
```

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

In [5]:
```
c = np.array([1,"abc"])
c
```

Out[5]:
```
array(['1', 'abc'], dtype='<U21')
```

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 [6]:
```
c = np.array([(0,1,0),[2,3,2]])
c
```

Out[6]:
```
array([[0, 1, 0],
       [2, 3, 2]])
```

In [7]:
```
d = np.array([(0,1,0),range(3)])
d
```

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

In [8]:
```
cproblem = np.array([(0,1),[2,3,2]])
```

```
/tmp/ipykernel_74734/1811570240.py:1: VisibleDeprecationWarning: Creating an ndarray from ragged nested sequences (which is
a list-or-tuple of lists-or-tuples-or ndarrays with different lengths or shapes) is deprecated. If you meant to do this, you
must specify 'dtype=object' when creating the ndarray.
  cproblem = np.array([(0,1),[2,3,2]])
```

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

In [9]:
```
d = np.array([[(0,1,0),[2,3,2]],[[4,5,4],[6,7,6]]])
d
```

Out[9]:
```
array([[[0, 1, 0],
        [2, 3, 2]],

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

Pointwise scalar operations (aka "broadcasting")

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

Out[10]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [11]:
```
a**3  # Replace each element by its cube
```

Out[11]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

In [12]:
```
a+3   # Add 3 to each element
```

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

In [13]:
```
3*a   # Multiply each element by 3
```

Out[13]:
```
array([ 0,  3,  6,  9, 12, 15, 18, 21, 24, 27])
```

In [14]:
```
3+a
```

Out[14]:
```
array([ 3,  4,  5,  6,  7,  8,  9, 10, 11, 12])
```

In [15]:
```
a-3, 3-a
```

Out[15]:
```
(array([-3, -2, -1,  0,  1,  2,  3,  4,  5,  6]),
 array([ 3,  2,  1,  0, -1, -2, -3, -4, -5, -6]))
```

In [16]:
```
a**3
```

Out[16]:
```
array([  0,   1,   8,  27,  64, 125, 216, 343, 512, 729])
```

- Not possible with standard Python lists

In [17]:
```
l = list(range(10))
```

In [18]:
```
3+l
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In [18], line 1
----> 1 3+l

TypeError: unsupported operand type(s) for +: 'int' and 'list'
```

- Can add two arrays pointwise, but also an array and a list of the same size

In [19]:
```
a+a, a+l  #Works!
```

Out[19]:
```
(array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]),
 array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18]))
```

Indexing and slicing

In [20]:
```
a[2], a[2:5]
```

Out[20]:
```
(2, array([2, 3, 4]))
```

Slice update

In [21]:
```
a[:6:2] = -1000  # equivalent to a[0:6:2] = -1000
a
```

Out[21]:
```
array([-1000,     1, -1000,     3, -1000,     5,     6,     7,     8,
           9])
```

- Unlike lists, slice update of array cannot resize the slice

In [22]:
```
a[2:5] = np.arange(2)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In [22], line 1
----> 1 a[2:5] = np.arange(2)

ValueError: could not broadcast input array from shape (2,) into shape (3,)
```

Populate an array from a function

- Index is implicitly used as argument to the function
- `b[i,j]` is `f(i,j)`

In [23]:
```
def f(x,y):
    return(10*x +  y)
```

In [24]:
```
f(5,7)
```

Out[24]:
```
57
```

In [25]:
```
b = np.fromfunction(f,(5,4),dtype=int)
b
```

Out[25]:
```
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
```

Indexing multi-dimensional arrays

In [26]:
```
b[2,3]  # Not b[2][3]
```

Out[26]:
```
23
```

In [27]:
```
b[0:5, 1] # second column in each row of b
```

Out[27]:
```
array([ 1, 11, 21, 31, 41])
```

In [28]:
```
b[ : ,1]  # equivalent to the previous example
```

Out[28]:
```
array([ 1, 11, 21, 31, 41])
```

In [29]:
```
b[1:3, :]  # all columns in the second and third row of b
```

Out[29]:
```
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

In [30]:
```
b[1:4,1:3] # extract a rectangular submatrix
```

Out[30]:
```
array([[11, 12],
       [21, 22],
       [31, 32]])
```

Iterating over elements

In [31]:
```
print(b)
```

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

In [32]:
```
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 [33]:
```
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
```

Arrays and types

- When we pass a sequence of values, type is taken from the values we pass
- Default is float64, 64-bit float
- `np.zeros()` creates a zero array of required dimensions, but what "type" of () do we get?

In [34]:
```
a = np.zeros((5,7))
a
```

Out[34]:
```
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.]])
```

If we want `int` zeroes, we pass the desired type as an argument

In [35]:
```
a = np.zeros((5,7),dtype=int)
a
```

Out[35]:
```
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 an array of random numbers
- Uniformly distributed in $[0, 1)$

In [36]:
```
a = np.random.random((5,7))
a
```

Out[36]:
```
array([[0.88213435, 0.16971307, 0.01241608, 0.76046308, 0.02927208,
        0.22720506, 0.28021178],
       [0.96245736, 0.89094659, 0.96283773, 0.76684503, 0.53365983,
        0.51587229, 0.08982928],
       [0.33130589, 0.52477119, 0.87205957, 0.33691265, 0.26971441,
        0.85020038, 0.90909779],
       [0.1431626 , 0.68307767, 0.48676025, 0.52794108, 0.68903994,
        0.80502854, 0.01381994],
       [0.73000794, 0.38163447, 0.55841497, 0.58133621, 0.73888899,
        0.36052911, 0.66850846]])
```

- Scale and shift to get desired range
- For instance, $[-5, 5)$

In [37]:
```
a = np.random.random((5,7))*10-5
a
```

Out[37]:
```
array([[ 4.96844914, -3.26404413,  3.56921996, -4.27737971, -2.39299866,
        -1.86906705, -0.34227678],
       [ 0.01990647,  4.65734971,  3.60619785, -4.67756946, -1.45417392,
        -3.94531739, -1.80933633],
       [ 0.74350601, -1.43335638, -1.89663478,  0.88976024, -4.10888142,
         4.01583694,  0.74279047],
       [-0.79795651,  0.10458986, -2.65581491,  2.96077438, -1.97117565,
        -4.97546063, -1.34820046],
       [ 1.51616659, -2.91947874,  4.1861112 ,  2.91955603,  3.1778907 ,
         0.96577325, -4.20250544]])
```

In [ ]:
```
```