NumpyBasic

November 30, 2023

NUMPY

Outline + Introduction + The basic of Numpy + Computation on Numpy (Universal Function) + Aggregation + Computation on Arrays Broadcasting + Comparisons, Masks and Boolean Logic + Fancy Indexing + Sorting Arrays

0.1 Introduction

[21]:
```
import numpy as np
```

0.1.1 Creating Array from python List

[20]:
```
z = np.array([[1, 2, 3, 4],[5, 6, 7, 9]])
```

[21]:
```
print(z)
```

```
[[1 2 3 4]
 [5 6 7 9]]
```

0.1.2 Creating array with list comprehension

[18]:
```
y = np.array([range(i, i + 3) for i in [2, 4, 6]])
```

[19]:
```
print(y)
```

```
[[2 3 4]
 [4 5 6]
 [6 7 8]]
```

0.1.3 Creating Array from Scratch

- scratch ~ zeros, ones, full, arange ...

[2]:
```
import numpy as np
```

[4]:
```
np.zeros(10, dtype=int)
```

[4]:
```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

[5]:
```
np.ones((3, 5), dtype=float)
```

[5]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

[6]:
```
np.full((3, 5), 3.14)
```

[6]:
```
array([[3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14],
       [3.14, 3.14, 3.14, 3.14, 3.14]])
```

[7]:
```
np.arange(0, 20, 2)
```

[7]:
```
array([ 0,  2,  4,  6,  8, 10, 12, 14, 16, 18])
```

[8]:
```
np.linspace(0, 1, 5)
```

[8]:
```
array([0.  , 0.25, 0.5 , 0.75, 1.  ])
```

[25]:
```
np.random.random((3, 3, 3))
```

[25]:
```
array([[[0.06734795, 0.55423645, 0.88918158],
        [0.44925373, 0.27265312, 0.18703768],
        [0.42334227, 0.08741301, 0.91625584]],

       [[0.29168978, 0.55712297, 0.97543155],
        [0.90094525, 0.63954351, 0.8265885 ],
        [0.2638245 , 0.99423442, 0.47500301]],

       [[0.13837376, 0.88474177, 0.40610085],
        [0.92259648, 0.70002807, 0.07683913],
        [0.4630417 , 0.71329737, 0.33119862]]])
```

[29]:
```
np.random.rand(3, 3)
```

[29]:
```
array([[0.42139513, 0.10759067, 0.96186905],
       [0.31489111, 0.71834156, 0.33289229],
       [0.75842387, 0.83351107, 0.16536809]])
```

[11]:
```
Normal = np.random.normal(0, 1, (3, 3))
Normal
```

[11]:
```
array([[ 0.16963058,  1.07033088,  1.04737355],
       [ 1.38843647, -0.72425817, -0.71990644],
       [ 0.72280092,  0.29706282,  0.1094254 ]])
```

[30]:
```
import scipy as sp
```

[28]:
```
Normal = np.random.normal(0, 1, (3, 3))
Normal
```

[28]:
```
array([[ 0.64108829,  0.37826722, -0.36737326],
       [ 0.74887206, -1.40438092,  0.40420644],
       [-0.99146478, -1.0909953 , -0.30652906]])
```

[13]:
```
np.mean(Normal), np.std(Normal)
```

[13]:
```
(-0.5284280817901283, 0.875746840142919)
```

[6]:
```
np.random.randint(0, 10, (3, 3))
```

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

[7]:
```
np.eye(3)
```

[7]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

[9]:
```
np.empty(3)
```

[9]:
```
array([1., 1., 1.])
```

0.2 The Basics of NumPy Arrays

0.2.1 NumPy Array Attributes

[21]:
```
np.random.seed(0) # seed for reproducibility
x1 = np.random.randint(10, size=6) # One-dimensional array
x2 = np.random.randint(10, size=(3, 4)) # Two-dimensional array
x3 = np.random.randint(10, size=(3, 4, 5)) # Three-dimensional array
```

[19]:
```
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
```

Three dimensional array: + x3 = np.random.randint(10, size=(3, 4, 5)) => 5 isany face amin'ny cube izay hita: 12 + 12 + 12 + 12 +12 = 60

[14]:
```
print(x2)
```

```
[[3 5 2 4]
 [7 6 8 8]
 [1 6 7 7]]
```

[15]:
```
x2
```

[15]:
```
array([[3, 5, 2, 4],
       [7, 6, 8, 8],
       [1, 6, 7, 7]])
```

[16]:
```
x1
```

[16]:
```
array([5, 0, 3, 3, 7, 9])
```

[18]:
```
print("dtype:", x3.dtype)
print("itemsize:", x3.itemsize, "bytes")
print("nbytes:", x3.nbytes, "bytes")
```

```
dtype: int32
itemsize: 4 bytes
nbytes: 240 bytes
```

0.2.2 Array Indexing: Accessing Single Elements

[5]:
```
x2[0, 0]
```

[5]:
```
3
```

[6]:
```
x1[0]
```

[6]:
```
5
```

[7]:
```
x1[-1]
```

[7]:
```
9
```

[9]:
```
x2[0, 0] = 12
print(x2)
```

```
[[12  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

[10]:
```
x1[0] = 3.14159 # this will be truncated!
print(x1)
```

```
[3 0 3 3 7 9]
```

0.2.3 Array Slicing: Accessing Subarrays

One-dimensional subarrays

[22]:
```
x = np.arange(10)
x
```

[22]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[23]:
```
x[:5]
```

[23]:
```
array([0, 1, 2, 3, 4])
```

[7]:
```
x[5:]
```

[7]:
```
array([5, 6, 7, 8, 9])
```

[17]:
```
x[4:7]
```

[17]:
```
array([4, 5, 6])
```

[18]:
```
x[::2]
```

[18]:
```
array([0, 2, 4, 6, 8])
```

[19]:
```
x[1::2]
```

[19]:
```
array([1, 3, 5, 7, 9])
```

[20]:
```
x[::-1]
```

[20]:
```
array([9, 8, 7, 6, 5, 4, 3, 2, 1, 0])
```

[23]:
```
x[5::-2]
```

[23]:
```
array([5, 3, 1])
```

Multi-dimensional subarrays

[8]:
```
x2[:2, :3]
```

[8]:
```
array([[3, 5, 2],
       [7, 6, 8]])
```

[25]:
```
x2[:3, ::2]
```

[25]:
```
array([[12,  2],
       [ 7,  8],
       [ 1,  7]])
```

[26]:
```
x2[::-1, ::-1]
```

[26]:
```
array([[ 7,  7,  6,  1],
       [ 8,  8,  6,  7],
       [ 4,  2,  5, 12]])
```

Accessing array rows and columns

[26]:
```
print(x2[:, 0]) # fakana vecteur colonne
```

```
[12  7  1]
```

[31]:
```
print(x2[0, :]) # fakana vecteur ligne
```

```
[12  5  2  4]
```

[33]:
```
print(x2[2])
```

```
[1 6 7 7]
```

Subarrays as no-copy views Let's extract a 2×2 subarray from this:

[30]:
```
x2_sub = x2[:2, :2]
print(x2_sub)
```

```
[[12  5]
 [ 7  6]]
```

[31]:
```
x2_sub[0, 0] = 99
print(x2_sub)
```

```
[[99  5]
 [ 7  6]]
```

[32]:
```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Creating copies of arrays

[33]:
```
x2_sub_copy = x2[:2, :2].copy()
```

[34]:
```
print(x2_sub_copy)
```

```
[[99  5]
 [ 7  6]]
```

[35]:
```
x2_sub_copy[0, 0] = 42
print(x2_sub_copy)
```

```
[[42  5]
 [ 7  6]]
```

[36]:
```
print(x2)
```

```
[[99  5  2  4]
 [ 7  6  8  8]
 [ 1  6  7  7]]
```

Reshaping of Arrays

[38]:
```
grid = np.arange(1, 10).reshape((3, 3))
```

[39]:
```
print(grid)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

[69]:
```
x = np.array([1, 2, 3])
x
```

[69]:
```
array([1, 2, 3])
```

[41]:
```
x.reshape((1, 3))
```

[41]:
```
array([[1, 2, 3]])
```

[70]:
```
x[np.newaxis, :]
```

[70]:
```
array([[1, 2, 3]])
```

- Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix. You can do this with the reshape method, **or** more easily by making use of the **newaxis keyword within a slice operation**:

[43]:
```
x.reshape((3, 1))
```

[43]:
```
array([[1],
       [2],
       [3]])
```

[44]:
```
x[:, np.newaxis]
```

[44]:
```
array([[1],
       [2],
       [3]])
```

0.2.4 Array Concatenation and Splitting

Concatenation, or **joining** of two arrays in NumPy, is primarily accomplished through the routines **np.concatenate, np.vstack, and np.hstack**.

Concatenation of arrays

- np.concatenate takes a tuple or list of arrays as its first argument

[10]:
```
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

[10]:
```
array([1, 2, 3, 3, 2, 1])
```

[6]:
```
z = [99, 99, 99]
print(np.concatenate([x, y, z]))
```

```
[ 1  2  3  3  2  1 99 99 99]
```

[55]:
```
grid = np.array([[1, 2, 3],
[4, 5, 6]])
```

[57]:
```
#np.concatenate([grid, grid])
```

- np.vstack

[36]:
```
x = np.array([1, 2, 3])
grid = np.array([[9, 8, 7],
[6, 5, 4]])
```

[37]:
```
np.vstack([x, grid])
```

[37]:
```
array([[1, 2, 3],
       [9, 8, 7],
       [6, 5, 4]])
```

- np.hstack

[38]:
```
y = np.array([[99],
[99]])
np.hstack([grid, y])
```

[38]:
```
array([[ 9,  8,  7, 99],
       [ 6,  5,  4, 99]])
```

Remark:

[66]:
```
print(np.concatenate([grid, grid2], axis=0)) #  axis = 0 ~ np.vstack
```

```
[[1 2 3]
 [4 5 6]
 [1 2 3]
 [4 5 6]]
```

[67]:
```
print(np.concatenate([grid, grid2], axis=1)) # axis = 1 ~ np.hstack
```

```
[[1 2 3 1 2 3]
 [4 5 6 4 5 6]]
```

[65]:
```
grid2 = np.array([[1, 2, 3],
[4, 5, 6]])
```

[ ]:
```
```

Splitting of arrays

[39]:
```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5]) # [3,5]=> 3 no anatiny x1 ka dimy halavan'ilay liste hatrany amin'ny x2 <=> 2 no isan'ny x2.
print(x1, x2, x3)
```

```
[1 2 3] [99 99] [3 2 1]
```

[19]:
```
grid = np.arange(16).reshape((4, 4))
grid
```

[19]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

[24]:
```
upper, lower = np.vsplit(grid, [2]) #ligne io [2] io eto
print(upper)
print(lower)
```

```
[[0 1 2 3]
 [4 5 6 7]]
[[ 8  9 10 11]
 [12 13 14 15]]
```

[25]:
```
left, right = np.hsplit(grid, [2]) #colonne io [2] io eto
print(left)
print(right)
```

```
[[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
[[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

[42]:
```
grid1 = np.arange(64).reshape((8, 8))
grid1
```

[42]:
```
array([[ 0,  1,  2,  3,  4,  5,  6,  7],
       [ 8,  9, 10, 11, 12, 13, 14, 15],
       [16, 17, 18, 19, 20, 21, 22, 23],
       [24, 25, 26, 27, 28, 29, 30, 31],
       [32, 33, 34, 35, 36, 37, 38, 39],
       [40, 41, 42, 43, 44, 45, 46, 47],
       [48, 49, 50, 51, 52, 53, 54, 55],
       [56, 57, 58, 59, 60, 61, 62, 63]])
```

[52]:
```
left1, right1 = np.hsplit(grid1, [2]) #colonne io [2] io eto
print(left1)
print(right1)
```

```
[[ 0  1]
 [ 8  9]
 [16 17]
 [24 25]
 [32 33]
 [40 41]
 [48 49]
 [56 57]]
[[ 2  3  4  5  6  7]
 [10 11 12 13 14 15]
 [18 19 20 21 22 23]
 [26 27 28 29 30 31]
 [34 35 36 37 38 39]
 [42 43 44 45 46 47]
 [50 51 52 53 54 55]
 [58 59 60 61 62 63]]
```

0.3 Computation on NumPy Arrays

0.3.1 Universal Function

Computation on NumPy arrays: Computation on NumPy arrays can be very fast, or it can be very slow. The key to making it fast is to use vectorized operations, generally implemented through Num- Py's universal functions (ufuncs). This section motivates the need for NumPy's ufuncs, which can be used to make repeated calculations on array elements much more effi- cient. It then introduces many of the most common and useful arithmetic ufuncs available in the NumPy package

- Ufuncs exist in two flavors: unary ufuncs, which operate on a single input, and binary ufuncs, which operate on two inputs. We'll see examples of both these types of functions here

Array arithmetic

[73]:
```
x = np.arange(4)
print("x =", x)
```

```
x = [0 1 2 3]
```

[27]:
```
print("x + 5 =", x + 5)
print("x - 5 =", x - 5)
print("x * 2 =", x * 2)
print("x / 2 =", x / 2)
print("x // 2 =", x // 2) # floor division
```

```
x + 5 = [5 6 7 8]
x - 5 = [-5 -4 -3 -2]
x * 2 = [0 2 4 6]
x / 2 = [0.  0.5 1.  1.5]
x // 2 = [0 0 1 1]
```

[28]:
```
print("-x = ", -x)
print("x ** 2 = ", x ** 2)
print("x % 2 = ", x % 2)
```

```
-x =  [ 0 -1 -2 -3]
x ** 2 =  [0 1 4 9]
x % 2 =  [0 1 0 1]
```

[77]:
```
-(0.5*x + 1)** 2 #strange in IT -a^n != a^n but (-a)^n = (a)^n. So, pay␣
↪attention :)
```

[77]:
```
array([-1.  , -2.25, -4.  , -6.25])
```

[16]:
```
np.add(x, 5)
```

[16]:
```
array([5, 6, 7, 8])
```

Absolute value

[34]:
```
x = np.array([-2, -1, 0, 1, 2])
abs(x)
```

[34]:
```
array([2, 1, 0, 1, 2])
```

[35]:
```
np.absolute(x)
```

[35]:
```
array([2, 1, 0, 1, 2])
```

[36]:
```
np.abs(x)
```

[36]:
```
array([2, 1, 0, 1, 2])
```

[17]:
```
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
print(np.abs(x)) #module no resultat havoakan'io
```

```
[5. 5. 2. 1.]
```

Trigonometric functions

[79]:
```
theta = np.linspace(0, np.pi, 5) # mizara 5 ny element ao anatiny arrays
print(theta)
```

```
[0.         0.78539816 1.57079633 2.35619449 3.14159265]
```

[42]:
```
print("theta = ", theta)
print("sin(theta) = ", np.sin(theta))
print("cos(theta) = ", np.cos(theta))
print("tan(theta) = ", np.tan(theta))
```

```
theta =  [0.         1.57079633 3.14159265]
sin(theta) =  [0.0000000e+00 1.0000000e+00 1.2246468e-16]
cos(theta) =  [ 1.000000e+00  6.123234e-17 -1.000000e+00]
tan(theta) =  [ 0.00000000e+00  1.63312394e+16 -1.22464680e-16]
```

Exponents and logarithms

[80]:
```
x = [1, 2, 3]
print("x =", x)
print("e^x =", np.exp(x))
print("2^x =", np.exp2(x))
print("3^x =", np.power(3, x))
```

```
x = [1, 2, 3]
e^x = [ 2.71828183  7.3890561  20.08553692]
2^x = [2. 4. 8.]
3^x = [ 3  9 27]
```

[18]:
```
x = [1, 2, 4, 10]
print("x =", x)
print("ln(x) =", np.log(x))
```

```
x = [1, 2, 4, 10]
ln(x) = [0.         0.69314718 1.38629436 2.30258509]
```

[48]:
```
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
log2(x) = [0.         1.         2.         3.32192809]
log10(x) = [0.         0.30103    0.60205999 1.        ]
```

[49]:
```
x = [0, 0.001, 0.01, 0.1]
print("exp(x) - 1 =", np.expm1(x))
print("log(1 + x) =", np.log1p(x))
```

```
exp(x) - 1 = [0.         0.0010005  0.01005017 0.10517092]
log(1 + x) = [0.         0.0009995  0.00995033 0.09531018]
```

Advanced Ufunc Features *Many NumPy users make use of ufuncs without ever learning their full set of features. We'll outline a few specialized features of ufuncs here.*

Specifying output For large calculations, it is sometimes useful to be able to specify the array where the result of the calculation will be stored. Rather than creating a temporary array, you can use this to write computation results directly to the memory location where you'd like them to be. For all ufuncs, you can do this using the out argument of the function:

[55]:
```
x = np.arange(5)
y = np.empty(5)
np.multiply(x, 10, out=y)
print(x)
print(y)
```

```
[0 1 2 3 4]
[ 0. 10. 20. 30. 40.]
```

[54]:
```
y = np.zeros(10)
np.power(2, x, out=y[::2])
print(y)
```

```
[ 1.  0.  2.  0.  4.  0.  8.  0. 16.  0.]
```

[56]:
```
y = np.zeros(10)
np.power(x, 2, out=y[::2])
print(y)
```

```
[ 0.  0.  1.  0.  4.  0.  9.  0. 16.  0.]
```

Aggregates

[57]:
```
x = np.arange(1, 6)
np.add.reduce(x)
```

[57]: 15

[58]:
```
np.multiply.reduce(x)
```

[58]: 120

[59]:
```
np.add.accumulate(x)
```

[59]: array([1, 3, 6, 10, 15])

[60]:
```
np.multiply.accumulate(x)
```

[60]: array([1,   2,   6, 24, 120])

Outer products

[82]:
```
#x = np.arange(1, 6)
#np.multiply.outer(x, x) # azo lazaina produit matrice
```

[88]:
```
x = np.arange(1, 6)
np.add.outer(x, x) # tsy mazava loatra
```

[88]:
```
array([[ 2,  3,  4,  5,  6],
       [ 3,  4,  5,  6,  7],
       [ 4,  5,  6,  7,  8],
       [ 5,  6,  7,  8,  9],
       [ 6,  7,  8,  9, 10]])
```

0.4 Aggregations: Min, Max, and Everything In Between

0.4.1 Summing the Values in an Array

[68]:
```
List = np.random.random(100)
print(List)
sum(List)
```

```
[0.14830987 0.42362661 0.41208317 0.79612958 0.35282562 0.22152099
 0.2243945  0.52339295 0.39423414 0.35275277 0.55613322 0.18804186
 0.36565539 0.62823454 0.72700235 0.58678367 0.87986117 0.98335057
 0.62086848 0.24442188 0.86593787 0.06554245 0.94331644 0.51506993
 0.38564522 0.6459047  0.81581069 0.51625443 0.0421467  0.22747895
 0.78202765 0.93805703 0.48536515 0.93504358 0.08751925 0.36478358
 0.37487491 0.13306103 0.02079136 0.13831275 0.32975248 0.87980124
 0.88846402 0.36956329 0.00120234 0.13934918 0.62153526 0.49709989
 0.04864059 0.81644637 0.65300874 0.09020387 0.77423007 0.16854508
 0.58794031 0.03367943 0.18499564 0.57623411 0.60113124 0.2571531
 0.5884187  0.15376836 0.51895392 0.85297688 0.44079571 0.12323286
 0.76124178 0.08257612 0.66311998 0.04229314 0.51933351 0.37251252
 0.02507405 0.61676684 0.02314291 0.59208653 0.289995   0.23431633
 0.24991211 0.83852454 0.48228092 0.99526543 0.14232845 0.76655617
 0.42253136 0.43968612 0.95450376 0.77082968 0.37094933 0.54897389
 0.67624041 0.96120476 0.62836751 0.97785562 0.91618682 0.47552354
 0.34989211 0.60950302 0.62955761 0.96166104]
```

[68]:
```
48.50048255401841
```

[69]:
```
len(List)
```

[69]:
```
100
```

[70]:
```
np.sum(List)
```

[70]:
```
48.50048255401839
```

[3]:
```
big_array = np.random.rand(1000000)
%timeit sum(big_array)
%timeit np.sum(big_array)
```

```
140 ms ± 7.45 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
2.52 ms ± 52.8 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

0.4.2 Minimum and Maximum

[72]:
```
min(big_array), max(big_array)
```

[72]:
```
(8.795922714632809e-07, 0.9999998958457029)
```

[73]:
```
np.min(big_array), np.max(big_array)
```

[73]:
```
(8.795922714632809e-07, 0.9999998958457029)
```

[74]:
```
%timeit min(big_array)
%timeit np.min(big_array)
```

```
73 ms ± 4.74 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
1.42 ms ± 37.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

[4]:
```
print(big_array.min(), big_array.max(), big_array.sum())
```

```
4.234815050851992e-07 0.9999998540881762 500209.65543838235
```

0.4.3 Multi dimensional aggregates

[5]:
```
M = np.random.random((3, 4))
print(M)
```

```
[[0.67883854 0.26126298 0.48634406 0.60080656]
 [0.56412036 0.54240267 0.34095407 0.11609548]
 [0.38528584 0.61963586 0.16488276 0.73877451]]
```

[6]:
```
M.sum()
```

[6]:
```
5.499403680429994
```

[7]:
```
M.min(axis=0) # Mitady min any anaty colonne
```

[7]:
```
array([0.38528584, 0.26126298, 0.16488276, 0.11609548])
```

[8]:
```
M.min(axis=1) # Mitady min any anaty ligne
```

[8]:
```
array([0.26126298, 0.11609548, 0.16488276])
```

[9]:
```
M.max(axis=0)
```

[9]:
```
array([0.67883854, 0.61963586, 0.48634406, 0.73877451])
```

[10]:
```
M.max(axis=1)
```

[10]:
```
array([0.67883854, 0.56412036, 0.73877451])
```

0.5 Computation on Arrays: Broadcasting

0.5.1 Introducing Broadcasting

[11]:
```
a = np.array([0, 1, 2])
b = np.array([5, 5, 5])
a + b
```

[11]:
```
array([5, 6, 7])
```

[12]:
```
a + 5
```

[12]:
```
array([5, 6, 7])
```

[13]:
```
M = np.ones((3, 3))
M
```

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

[14]:
```
M + a
```

[14]:
```
array([[1., 2., 3.],
       [1., 2., 3.],
       [1., 2., 3.]])
```

[15]:
```
a = np.arange(3)
b = np.arange(3)[:, np.newaxis]
print(a)
print(b)
```

```
[0 1 2]
[[0]
 [1]
 [2]]
```

[16]:
```
a + b
```

[16]:
```
array([[0, 1, 2],
       [1, 2, 3],
       [2, 3, 4]])
```

0.5.2 Rules of Broadcasting

- Rule 1: If the two arrays differ in their number of dimensions, the shape of the one with fewer dimensions is padded with ones on its leading (left) side.

- Rule 2: If the shape of the two arrays does not match in any dimension, the array with shape equal to 1 in that dimension is stretched to match the other shape.
- Rule 3: If in any dimension the sizes disagree and neither is equal to 1, an error is raised.

[3]:
```
M = np.ones((2, 3))
a = np.arange(3)
```

Broadcasting example 1

[5]:
```
M + a
```

[5]:
```
array([[1., 2., 3.],
       [1., 2., 3.]])
```

Broadcasting example 2

[4]:
```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
a + b
```

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

Broadcasting example 3

[20]:
```
M = np.ones((3, 2))
a = np.arange(3)
#M + a
a[:, np.newaxis].shape
```

[20]:
```
(3, 1)
```

[21]:
```
M + a[:, np.newaxis]
```

[21]:
```
array([[1., 1.],
       [2., 2.],
       [3., 3.]])
```

[7]:
```
np.logaddexp(M, a[:, np.newaxis])
```

[7]:
```
array([[1.31326169, 1.31326169],
       [1.69314718, 1.69314718],
       [2.31326169, 2.31326169]])
```

0.5.3 Broadcasting in Practice

Centering an array

[6]:
```
X = np.random.random((10, 3))
X
```

[6]:
```
array([[0.57241121, 0.35546209, 0.25401201],
       [0.36658721, 0.49538061, 0.06982437],
       [0.4182157 , 0.12105542, 0.99623009],
       [0.51673222, 0.52796046, 0.64021156],
       [0.19065645, 0.98461454, 0.41072164],
       [0.56005914, 0.44680262, 0.7442828 ],
       [0.4762633 , 0.3022721 , 0.97395656],
       [0.59430637, 0.98151474, 0.32399177],
       [0.80944464, 0.33032092, 0.65013809],
       [0.56522769, 0.81711665, 0.55335373]])
```

[9]:
```
Xmean = X.mean(0) # 0 COLUMN
Xmean
```

[9]:
```
array([0.47482836, 0.62316541, 0.51364258])
```

[12]:
```
XMAX = X.max(0)
XMAX
```

[12]:
```
array([0.97860295, 0.90114393, 0.80042685])
```

[13]:
```
Xmean1 = X.mean(1) # 1 LIGNE
Xmean1
```

[13]:
```
array([0.52513912, 0.55219876, 0.34304109, 0.55627749, 0.38770347,
       0.26158176, 0.61255154, 0.85149666, 0.45433528, 0.53167097])
```

[14]:
```
X_centered = X - Xmean
X_centered.mean(0)
```

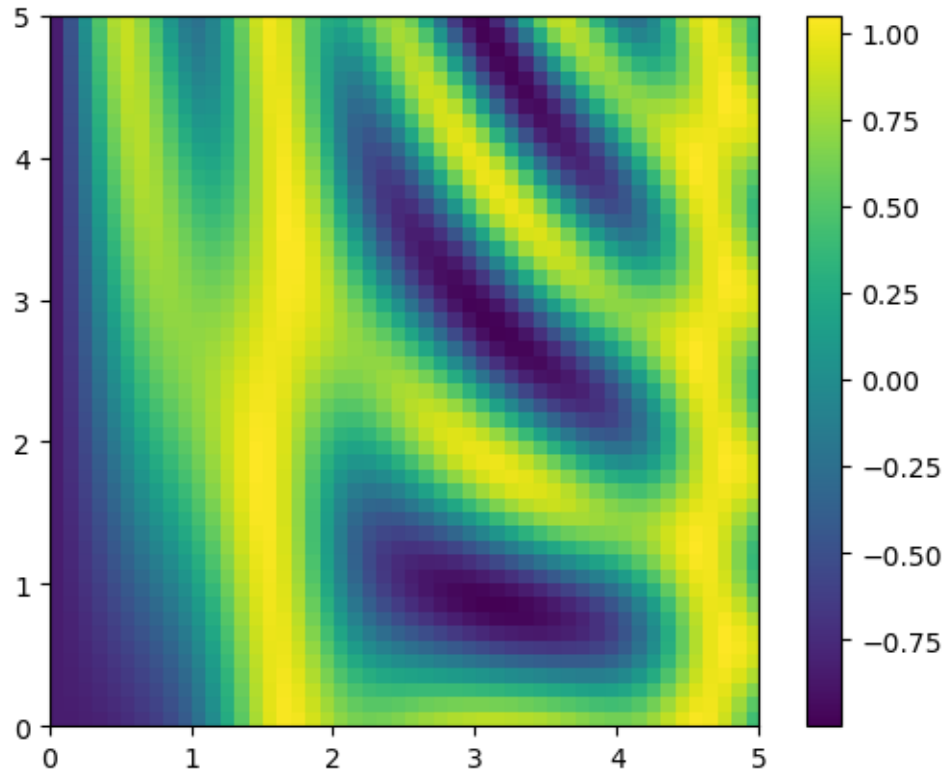
[14]:
```
array([ 0.02224959, -0.08888053, -0.02220657])
```

Plotting a two-dimensional function

[15]:
```
# x and y have 50 steps from 0 to 5
x = np.linspace(0, 5, 50)
y = np.linspace(0, 5, 50)[:, np.newaxis]
z = np.sin(x) ** 10 + np.cos(10 + y * x) * np.cos(x)
```

[16]:
```
%matplotlib inline
import matplotlib.pyplot as plt
```

[17]:
```
plt.imshow(z, origin='lower', extent=[0, 5, 0, 5],
cmap='viridis')
plt.colorbar();
```



0.6 Comparisons, Masks, and Boolean Logic

[8]:
```
rng = np.random.RandomState(0)
x = rng.randint(10, size=(3, 4))
```

0.6.1 Working with Boolean Arrays

[9]:
```
print(x)
```

```
[[5 0 3 3]
 [7 9 3 5]
 [2 4 7 6]]
```

0.6.2 Counting entries

[12]:
```
np.count_nonzero(x < 6)
```

[12]: 8

[13]:
```
np.sum(x < 6)
```

[13]: 8

- .count_nonzero() ~ .sum()

[15]:
```
np.sum(x < 6, axis=1) # how many values less than 6 in each row?
```

[15]:
```
array([4, 2, 2])
```

The benefit of sum() is that like with other NumPy aggregation functions, this sum- mation can be done along rows or columns

[29]:
```
np.any(x > 8)
```

[29]:
```
True
```

[30]:
```
np.any(x < 0)
```

[30]:
```
False
```

[31]:
```
np.all(x < 10)
```

[31]:
```
True
```

[32]:
```
np.all(x == 6)
```

[32]:
```
False
```

[33]:
```
np.all(x < 8, axis=1)
```

[33]:
```
array([ True, False,  True])
```

If we're interested in quickly checking whether any or all the values are true, we can use (you guessed it) np.any() or np.all()

0.6.3 Boolean operators

[27]:
```
x
```

[27]:
```
array([[5, 0, 3, 3],
       [7, 9, 3, 5],
       [2, 4, 7, 6]])
```

[31]:
```
x < 5
```

[31]:
```
array([[False,  True,  True,  True],
       [False, False,  True, False],
       [ True,  True, False, False]])
```

- Now to select these values from the array, we can simply index on this Boolean array; this is known as a masking operation.

[32]:
```
z = x[x < 5]
```

[33]:
```
print(z);
```

```
[0 3 3 3 2 4]
```

[ ]:
```

```

0.6.4 Using the Keywords and/or Versus the Operators &/|

[17]:
```
bool(42 and 0)
```

[17]:
```
False
```

[24]:
```
bool(42 or 0)
```

[24]:
```
True
```

[19]:
```
bin(42)
```

[19]:
```
'0b101010'
```

[12]:
```
bin(59)
```

[12]:
```
'0b111011'
```

[20]:
```
bin(42 & 59)
```

[20]:
```
'0b101010'
```

[26]:
```
bin(42 or 59)
```

[26]:
```
'0b101010'
```

[13]:
```
bin(42 | 59)
```

[13]:
```
'0b111011'
```

[21]:
```
A = np.array([1, 0, 1, 0, 1, 0], dtype=bool)
B = np.array([1, 1, 1, 0, 1, 1], dtype=bool)
A | B
```

[21]:
```
array([ True,  True,  True, False,  True,  True])
```

[22]:
```
A and B
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [22], in <cell line: 1>()
----> 1 A and B
```

```
ValueError: The truth value of an array with more than one element is ambiguous
  ↪Use a.any() or a.all()
```

When doing a Boolean expression on a given array, you should use | or & rather than or or and

[35]:
```
x = np.arange(10)
(x > 4) & (x < 8)
```

[35]:
```
array([False, False, False, False, False,  True,  True,  True, False,
       False])
```

[36]:
```
(x > 4) and (x < 8)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Input In [36], in <cell line: 1>()
----> 1 (x > 4) and (x < 8)

ValueError: The truth value of an array with more than one element is ambiguous
  ↪Use a.any() or a.all()
```

0.7 Fancy Indexing ~ subarray

Fancy indexing is conceptually simple: it means passing an array of indices to access multiple array elements at once.

0.7.1 Exploring Fancy Indexing

[109]:
```
import numpy as np
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
print(x)
```

```
[51 92 14 71 60 20 82 86 74 74]
```

[113]:
```
ind = [3, 7, 4]
a=x[ind]
```

[114]:
```
a
```

[114]:
```
array([71, 86, 60])
```

[116]:
```
x[ind]
```

[116]:
```
array([71, 86, 60])
```

[117]:
```
x
```

[117]:
```
array([51, 92, 14, 71, 60, 20, 82, 86, 74, 74])
```

[118]:
```
ind = np.array([[3, 7],
[4, 5]])
x[ind]
```

[118]:
```
array([[71, 86],
       [60, 20]])
```

[121]:
```
X = np.arange(12).reshape((3, 4))
X
```

[121]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

[122]:
```
row = np.array([0, 1, 2])
col = np.array([2, 1, 3])
X[row, col]
```

[122]:
```
array([ 2,  5, 11])
```

[124]:
```
X[row[:, np.newaxis], col] # nazava  tamin'ny 30 novembre 14h25
```

[124]:
```
array([[ 2,  1,  3],
       [ 6,  5,  7],
       [10,  9, 11]])
```

[123]:
```
row[:, np.newaxis]
```

[123]:
```
array([[0],
       [1],
       [2]])
```

[30]:
```
row[:, np.newaxis] * col
```

[30]:
```
array([[0, 0, 0],
       [2, 1, 3],
       [4, 2, 6]])
```

0.7.2 Combined Indexing

[31]:
```
print(X)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
```

[32]:
```
X[1:, [2, 0, 1]]
```

[32]:
```
array([[ 6,  4,  5],
       [10,  8,  9]])
```

[33]:
```
mask = np.array([1, 0, 1, 0], dtype=bool)
X[row[:, np.newaxis], mask]
```

[33]:
```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
```

0.7.3 Example: selecting Random Points

One common use of fancy indexing is the selection of subsets of rows from a matrix. For example, we might have an N by D matrix representing N points in D dimensions, such as the following points drawn from a two-dimensional normal distribution:

[34]:
```
mean = [0, 0]
cov = [[1, 2],
[2, 5]]
X = rand.multivariate_normal(mean, cov, 100)
X.shape
```

[34]:
```
(100, 2)
```

[56]:
```
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # for plot styling
plt.scatter(X[:, 0], X[:, 1]);
```

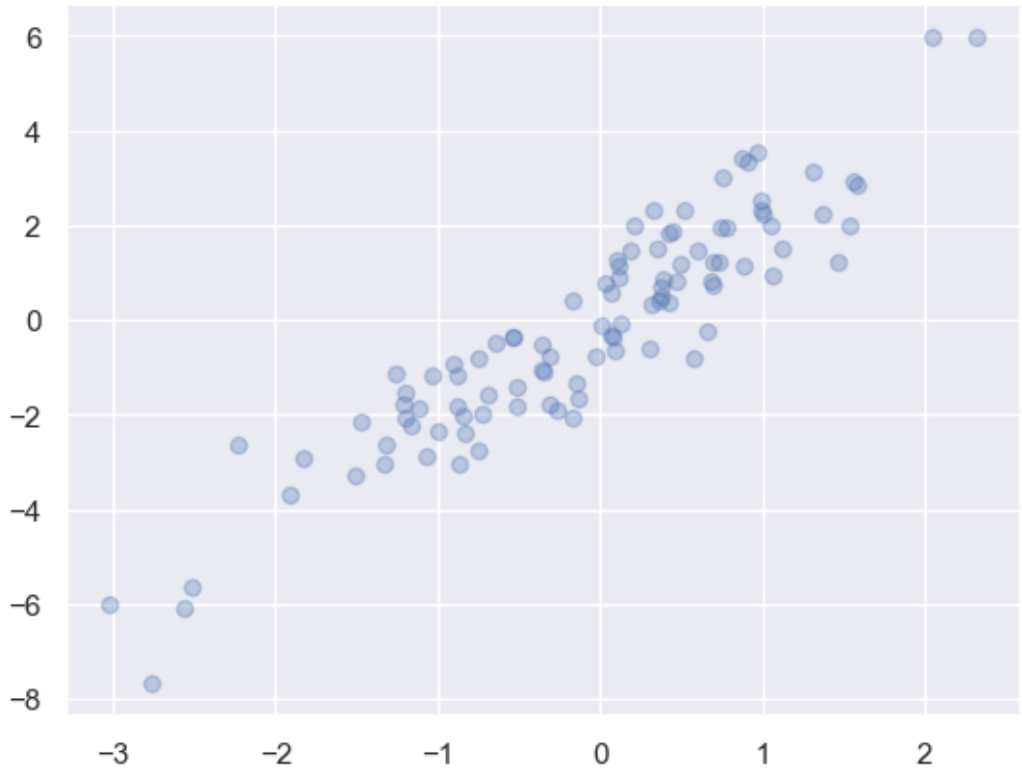
[36]:
```
indices = np.random.choice(X.shape[0], 20, replace=False)
indices
```

[36]:
```
array([11, 93, 26, 49, 14, 88,  8, 69, 46, 47, 12, 98, 53, 62, 84, 44, 24,
       22, 61, 94])
```

[37]:
```
selection = X[indices] # fancy indexing here
selection.shape
```

[37]:
```
(20, 2)
```

[40]:
```
plt.scatter(X[:, 0], X[:, 1], alpha=0.3)
plt.scatter(selection[:, 0], selection[:, 1], facecolor='none', s=200);
```



0.7.4 Modifying Values with Fancy Indexing

[41]:
```
x = np.arange(10)
i = np.array([2, 1, 8, 4])
x[i] = 99
print(x)
```

```
[ 0 99 99  3 99  5  6  7 99  9]
```

[42]:
```
x[i] -= 10
print(x)
```

```
[ 0 89 89  3 89  5  6  7 89  9]
```

[43]:
```
x = np.zeros(10)
x[[0, 0]] = [4, 6]
print(x)
```

```
[6. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
```

[44]:
```
i = [2, 3, 3, 4, 4, 4]
x[i] += 1
x
```

[44]:
```
array([6., 0., 1., 1., 1., 0., 0., 0., 0., 0.])
```

[45]:
```
x = np.zeros(10)
np.add.at(x, i, 1)
print(x)
```

```
[0. 0. 1. 2. 3. 0. 0. 0. 0. 0.]
```

0.7.5 Example: Binning Data

[126]:
```
np.random.seed(42)
x = np.random.randn(100)
```

[127]:
```
bins = np.linspace(-5, 5, 20)
counts = np.zeros_like(bins)
```

[131]:
```
bins
```

[131]:
```
array([-5.        , -4.47368421, -3.94736842, -3.42105263, -2.89473684,
       -2.36842105, -1.84210526, -1.31578947, -0.78947368, -0.26315789,
        0.26315789,  0.78947368,  1.31578947,  1.84210526,  2.36842105,
        2.89473684,  3.42105263,  3.94736842,  4.47368421,  5.        ])
```

[128]:
```
counts
```

[128]:
```
array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
       0., 0., 0.])
```

[129]:
```
i = np.searchsorted(bins, x)
```

[130]:
```
i
```

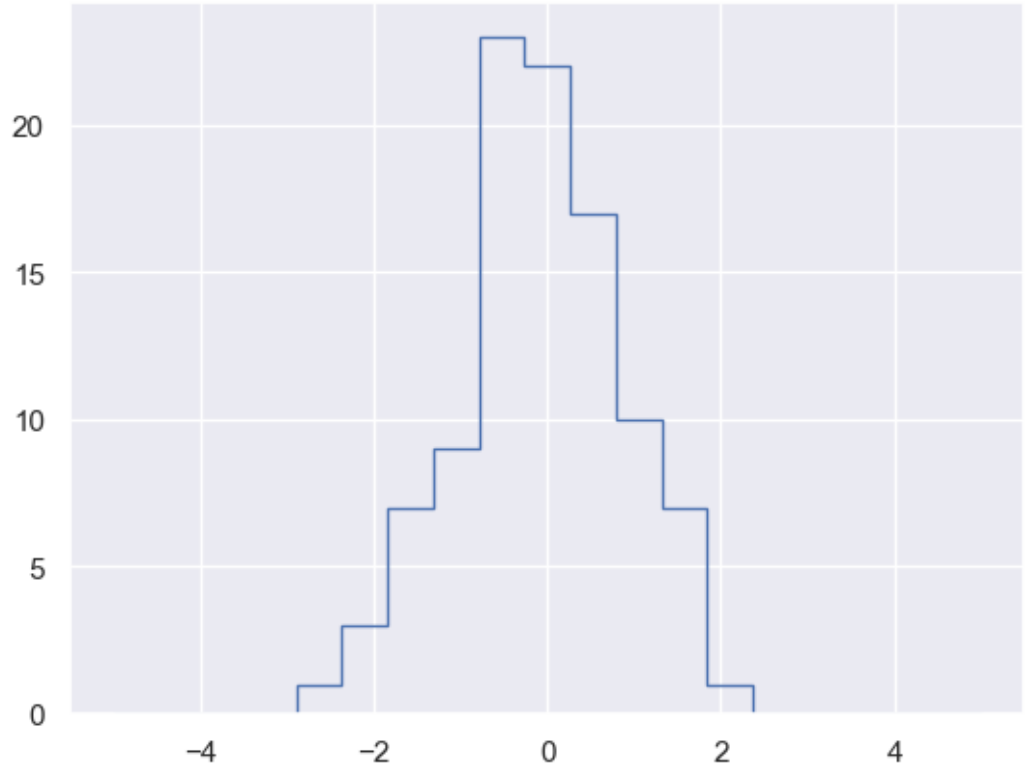
[130]:
```
array([11, 10, 11, 13, 10, 10, 13, 11,  9, 11,  9,  9, 10,  6,  7,  9,  8,
       11,  8,  7, 13, 10, 10,  7,  9, 10,  8, 11,  9,  9,  9, 14, 10,  8,
       12,  8, 10,  6,  7, 10, 11, 10, 10,  9,  7,  9,  9, 12, 11,  7, 11,
        9,  9, 11, 12, 12,  8,  9, 11, 12,  9, 10,  8,  8, 12, 13, 10, 12,
       11,  9, 11, 13, 10, 13,  5, 12, 10,  9, 10,  6, 10, 11, 13,  9,  8,
        9, 12, 11,  9, 11, 10, 12,  9,  9,  9,  7, 11, 10, 10, 10],
      dtype=int64)
```

[54]:
```
np.add.at(counts, i, 1)
```

[60]:
```
plt.hist(x, bins, histtype='step')
```

[60]:
```
(array([ 0.,  0.,  0.,  0.,  1.,  3.,  7.,  9., 23., 22., 17., 10.,  7.,
         1.,  0.,  0.,  0.,  0.,  0.]),
 array([-5.        , -4.47368421, -3.94736842, -3.42105263, -2.89473684,
        -2.36842105, -1.84210526, -1.31578947, -0.78947368, -0.26315789,
```

```
        0.26315789,  0.78947368,  1.31578947,  1.84210526,  2.36842105,
        2.89473684,  3.42105263,  3.94736842,  4.47368421,  5.        ]),
[<matplotlib.patches.Polygon at 0x249988daf10>])
```



[61]:
```
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)

print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
68.4 µs ± 3.74 µs per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
Custom routine:
32 µs ± 167 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

[62]:
```
x = np.random.randn(1000000)
print("NumPy routine:")
%timeit counts, edges = np.histogram(x, bins)
print("Custom routine:")
%timeit np.add.at(counts, np.searchsorted(bins, x), 1)
```

```
NumPy routine:
103 ms ± 25.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
Custom routine:
191 ms ± 1.86 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

[68]:
```
def selection_sort(x):
    for i in range(len(x)):
        swap = i + np.argmin(x[i:])
        (x[i], x[swap]) = (x[swap], x[i])
    return x
```

[69]:
```
x = np.array([2, 1, 4, 3, 5])
selection_sort(x)
```

[69]:
```
array([1, 2, 3, 4, 5])
```

[71]:
```
def bogosort(x):
    while np.any(x[:-1] > x[1:]):
        np.random.shuffle(x)
    return x
```

[72]:
```
x = np.array([2, 1, 4, 3, 5])
bogosort(x)
```

[72]:
```
array([1, 2, 3, 4, 5])
```

0.7.6 Fast Sorting in NumPy: np.sort and np.argsort

[73]:
```
x = np.array([2, 1, 4, 3, 5])
np.sort(x)
```

[73]:
```
array([1, 2, 3, 4, 5])
```

[74]:
```
x.sort()
print(x)
```

```
[1 2 3 4 5]
```

[132]:
```
x = np.array([2, 1, 4, 3, 5])
i = np.argsort(x) # A related function is argsort, which instead returns the␣
 ↪indices of the sorted
#elements:
print(i)
```

```
[1 0 3 2 4]
```

[76]:
```
x[i]
```

[76]:
```
array([1, 2, 3, 4, 5])
```

0.7.7 Sorting along rows or columns

[77]:
```
rand = np.random.RandomState(42)
X = rand.randint(0, 10, (4, 6))
print(X)
```

```
[[6 3 7 4 6 9]
 [2 6 7 4 3 7]
 [7 2 5 4 1 7]
 [5 1 4 0 9 5]]
```

[78]:
```
np.sort(X, axis=0)
```

[78]:
```
array([[2, 1, 4, 0, 1, 5],
       [5, 2, 5, 4, 3, 7],
       [6, 3, 7, 4, 6, 7],
       [7, 6, 7, 4, 9, 9]])
```

[79]:
```
np.sort(X, axis=1)
```

[79]:
```
array([[3, 4, 6, 6, 7, 9],
       [2, 3, 4, 6, 7, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 5, 9]])
```

0.7.8 Partial Sorts: Partitioning

[166]:
```
x = np.array([7, 2, 3, 1, 6, 5, 4, 40, 0, 343]) # mazava tamin'ny 30 novembre␣
↪14h47
np.partition(x, 3)
```

[166]:
```
array([  0,   1,   2,   3,   6,   5,   4,   7,  40, 343])
```

Sometimes we're not interested in sorting the entire array, but simply want to find the K smallest values in the array. NumPy provides this in the np.partition function. np.partition takes an array and a number K; the result is a new array with the smallest K values to the left of the partition, and the remaining values to the right, in arbitrary order

[91]:
```
np.partition(X, 2, axis=1)
```

[91]:
```
array([[3, 4, 6, 7, 6, 9],
       [2, 3, 4, 7, 6, 7],
       [1, 2, 4, 5, 7, 7],
       [0, 1, 4, 5, 9, 5]])
```

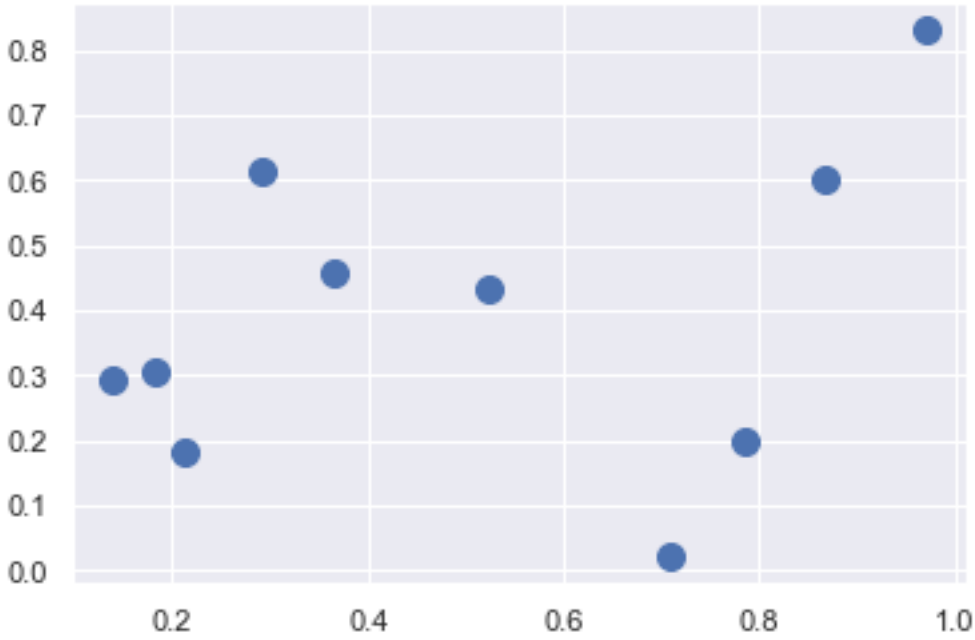
0.7.9 Example: k-Nearest Neighbors

Let's quickly see how we might use this argsort function along multiple axes to find the nearest neighbors of each point in a set. We'll start by creating a random set of 10 points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

[167]:
```
X = rand.rand(10, 2)
```

To get an idea of how these points look, let's quickly scatter plot them

[169]:
```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn; seaborn.set() # Plot styling
plt.scatter(X[:, 0], X[:, 1], s=100);
```



Now we'll compute the distance between each pair of points. Recall that the squareddistance between two points is the sum of the squared differences in each dimension; using the efficient broadcasting and aggregation ("Aggregations: Min, Max, and Everything in Between" on page 58) routines provided by NumPy, we can compute the matrix of square distances in a single line of code:

[170]:
```
dist_sq = np.sum((X[:,np.newaxis,:] - X[np.newaxis,:,:]) ** 2, axis=-1)
```

[173]:
```
differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
## for each pair of points, compute differences in their coordinates
differences.shape
```

[173]:
```
(10, 10, 2)
```

[174]:
```
# square the coordinate differences
sq_differences = differences ** 2
sq_differences.shape
```

[174]: (10, 10, 2)

[175]:
```
# sum the coordinate differences to get the squared distance
dist_sq = sq_differences.sum(-1)
dist_sq.shape
```

[175]: (10, 10)

Just to double-check what we are doing, we should see that the diagonal of this matrix (i.e., the set of distances between each point and itself) is all zero:

[176]:
```
dist_sq.diagonal()
```

[176]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

[187]:
```
nearest = np.argsort(dist_sq, axis=1)
print(nearest)
```

```
[[0 2 5 9 8 6 1 4 3 7]
 [1 9 5 3 8 4 0 7 6 2]
 [2 0 5 9 8 6 1 4 7 3]
 [3 4 7 8 5 6 1 9 0 2]
 [4 7 3 8 6 5 1 9 0 2]
 [5 8 6 9 4 0 3 7 1 2]
 [6 8 5 4 7 3 0 9 2 1]
 [7 4 3 8 6 5 1 9 0 2]
 [8 5 6 4 7 3 9 0 1 2]
 [9 1 5 0 8 3 4 6 7 2]]
```

Notice that the first column gives the numbers 0 through 9 in order: this is due to the fact that each point's closest neighbor is itself, as we would expect. By using a full sort here, we've actually done more work than we need to in this case. If we're simply interested in the nearest k neighbors, all we need is to partition each row so that the smallest k + 1 squared distances come first, with larger distances filling the remaining positions of the array. We can do this with the np.argpartition function:

[185]:
```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
```

[186]:
```
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
plt.scatter(X[:, 0], X[:, 1], s=100)

# draw lines from each point to its two nearest neighbors
K = 2
for i in range(X.shape[0]):
 for j in nearest_partition[i, :K+1]:
 # plot a line from X[i] to X[j]
 # use some zip magic to make it happen:
```

```
  plt.plot(*zip(X[j], X[i]), color='black')
```