In [1]:

```
# @gyleodhis=====gyleodhis@outlook.com=====
import numpy as np

# np.__version__ Checks numpy vesion
# Generate an array of 10 random integers less than 100
rand = np.random.RandomState(42)
x = rand.randint(100, size=10)
x
```

Out[1]:

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

Remember that unlike Python lists, NumPy is constrained to arrays that all contain the same type. If types do not match, NumPy will upcast if possible as below:

In [2]:

```
L=np.array([3.14, 4, 2, 3])
L
```

Out[2]:

```
array([3.14, 4.  , 2.  , 3.  ])
```

In [3]:

```
##If we want to explicitly set the data type of the resulting array, we can use the
L=np.array([1, 2, 3, 4], dtype='float32')
L
```

Out[3]:

```
array([1., 2., 3., 4.], dtype=float32)
```

Accesiing four different elemets in a array

In [4]:

```
k=[x[2], x[5], x[9], x[0]]
k
#We can also do it this way
z=[2,5,9,1]
x[z]
```

Out[4]:

```
array([14, 20, 74, 92])
```

In [5]:

```
#Multidimentional Array.
L=np.array([range(i, i + 3) for i in [2, 4, 6]])
L
```

Out[5]:

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

In [6]:

```
# Create a length-10 integer array filled with zeros
L=np.zeros(10, dtype=int)
L
```

Out[6]:

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

In [7]:

```
# Create a 3x5 floating-point array filled with 1s
L=np.ones((3, 5), dtype=float)
L
```

Out[7]:

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

In [8]:

```
# Create a 3x5 array filled with 3.14
L=np.full((3, 5), 3.14)
L
```

Out[8]:

```
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]])
```

In [9]:

```
# Create an array filled with a linear sequence
# Starting at 0, ending at 20, stepping by 2
# (this is similar to the built-in range() function)
L=np.arange(0, 20, 2)
L
```

Out[9]:

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

In [10]:

```
# Create an array of five values evenly spaced between 0 and 1
np.linspace(0, 1, 5)
```

Out[10]:

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

In [11]:

```
# Create a 3x3 array of uniformly distributed
# random values between 0 and 1
L=np.random.random((3, 3))
L
```

Out[11]:

```
array([[0.72513106, 0.76197945, 0.83390154],
       [0.54045859, 0.27981842, 0.41010531],
       [0.01503548, 0.91073155, 0.38098201]])
```

In [12]:

```
# Create a 3x3 array of normally distributed random values
# with mean 0 and standard deviation 1
np.random.normal(0, 1, (3, 3))
```

Out[12]:

```
array([[ 1.27034312, -0.65701202, -1.58951515],
       [ 0.34042173, -0.51818355, -0.61172979],
       [-0.98050539,  0.06304067,  1.32297523]])
```

In [13]:

```
# Create a 3x3 array of random integers in the interval [0, 10)
np.random.randint(0, 10, (3, 3))
```

Out[13]:

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

In [14]:

```
# Create a 3x3 identity matrix
np.eye(3)
```

Out[14]:

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

In [15]:

```
# Create an uninitialized array of three integers
# The values will be whatever happens to already exist at that
# memory location
np.empty(3)
```

Out[15]:

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

In [16]:

```
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
#Each array has attributes ndim (the number of dimensions), shape (the size of each
print("x3 ndim: ", x3.ndim)
print("x3 shape:", x3.shape)
print("x3 size: ", x3.size)
print("x3 type: ", x3.dtype)
print("x3 type: ", x3.itemsize)
```

```
x3 ndim:  3
x3 shape: (3, 4, 5)
x3 size:  60
x3 type:  int64
x3 type:  8
```

In [17]:

```
# Array Indexing: Accessing Single Elements
print(x1[4])
# To index from the end of the array, you can use negative indices:
print(x1[-1])
```

```
6
8
```

In [18]:

```
# In a multidimensional array, you access items using a comma-separated tuple of in
x2[2, 0]
```

Out[18]:

```
1
```

In [19]:

```
# You can also modify values using any of the above index notation:
x2[0, 0] = 12
x2[0,0]
```

Out[19]:

```
12
```

Keep in mind that, unlike Python lists, NumPy arrays have a fixed type. This means,for example, that if you attempt to insert a floating-point value to an integer array, the value will be silently truncated. Don't be caught unaware by this behavior!

x1[0] = 3.14159 # this will be truncated to 3!

Fancy indexing also works in multiple dimensions. Consider the following array:

In [20]:

```
x = np.arange(12).reshape((3,4))
x
```

Out[20]:

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

Like with standard indexing, the first index refers to the row, and the second to the column:

In [21]:

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

Out[21]:

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

Selecting Random Points.

In [22]:

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

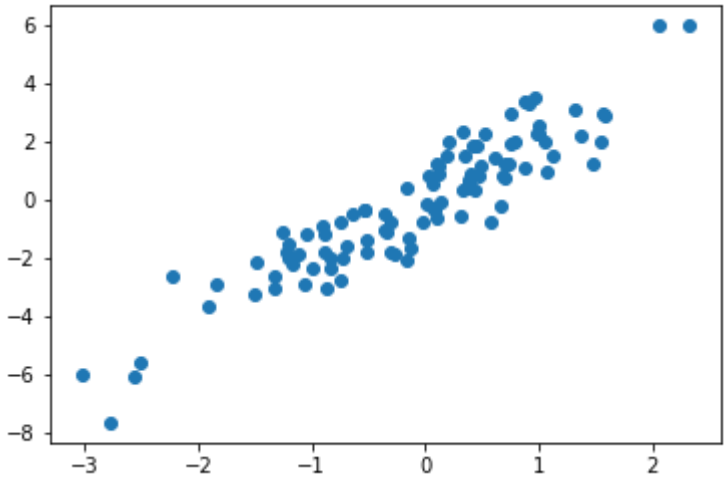
Out[22]:

```
(100, 2)
```

we can visualize these points as a scatter plot

In [23]:

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



Let's use fancy indexing to select 20 random points. We'll do this by first choosing 20 random indices with no repeats, and use these indices to select a portion of the origi- nal array:

In [24]:

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

Out[24]:

```
array([ 8, 58, 37, 49,  6, 42, 14,  7,  3,  9, 67, 22, 86, 84, 56, 9
4,  0,
       65, 21, 28])
```

In [25]:

```
selection = x[indices] # Fancy indexing here
selection.shape
```

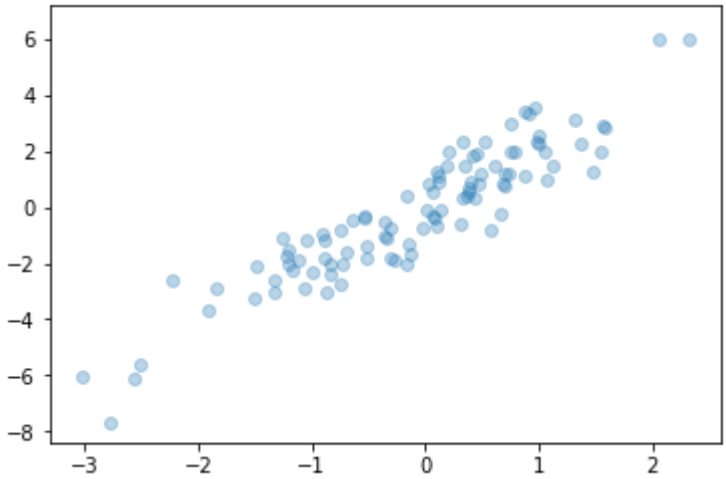
Out[25]:

```
(20, 2)
```

Now to see which points were selected, let's over-plot large circles at the locations of the selected points

In [26]:

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



This sort of strategy is often used to quickly partition datasets, as is often needed in train/test splitting for validation of statistical models

Modifying Values with Fancy Indexing. Just as fancy indexing can be used to access parts of an array, it can also be used to modify parts of an array

In [27]:

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

Out[27]:

```
array([ 0, 99, 99,  3, 99,  5,  6,  7, 99,  9])
```

In [28]:

```
i
```

Out[28]:

```
array([2, 1, 8, 4])
```

We can use any assignment-type operator for this. For example:

In [29]:

```
x[i] +=20
x
```

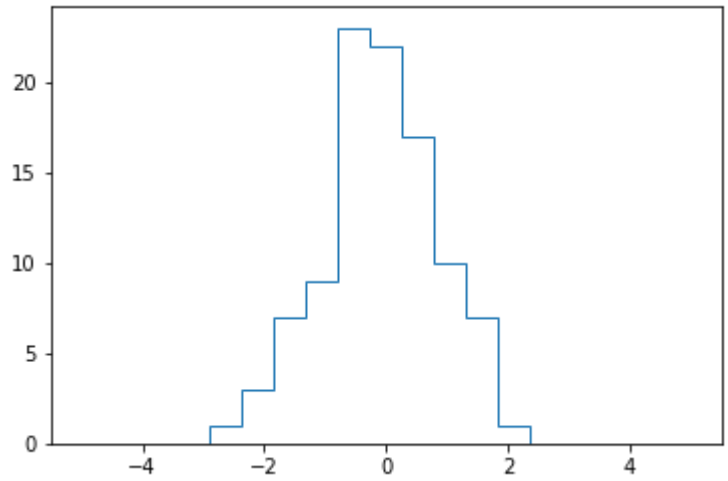
Out[29]:

```
array([  0, 119, 119,   3, 119,   5,   6,   7, 119,   9])
```

imagine we have 1,000 values and would like to quickly find where they fall within an array of bins. We could compute it using ufunc.at like this:

In [30]:

```
np.random.seed(42)
x=np.random.randn(100)
#compute a histogram by hand
bins = np.linspace(-5,5,20)
counts= np.zeros_like(bins)
# find the appropriate bin for each x
i=np.searchsorted(bins,x)
# add 1 to each of these bins
np.add.at(counts, i, 1)
#The counts now reflect the number of points within each bin—in other words, a hist
#plt.plot(bins, counts, linestyle='steps');
plt.hist(x, bins, histtype='step');
```



Fast sorting pf arrays with np.sort and np.argsort. To return a sorted version of the array without modifying the input, you can use np.sort:

In [31]:

```
x = np.array([2, 1, 4, 3, 5])
np.sort(x) #x.sort() produces the same result.
```

Out[31]:

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

argsort,returns the indices of the sorted elements:

In [32]:

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

Out[32]:

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

Sorting along rows or columns

In [33]:

```
rand = np.random.RandomState(42)
x = rand.randint(0, 10, (4, 6))
x
```

Out[33]:

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

In [34]:

```
# sort each column of X. Replace axix=0 with 1 to sort rows
np.sort(x, axis=0)
```

Out[34]:

```
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]])
```

Sometimes we're not interested in sorting the entire array, but simply want to find the K smallest values in the array.

In [35]:

```
x = np.array([7, 2, 3, 1, 6, 5, 4])
np.partition(x,4)
```

Out[35]:

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

Similarly to sorting, we can partition along an arbitrary axis of a multidimensional array:

In [36]:

```
np.partition(x, 2, axis=0)
```

Out[36]:

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

The result is an array where the first two slots in each row contain the smallest values from that row, with the remaining values filling the remaining slots.

Example: k-Nearest Neighbors. We will start by creating random set of 10 points on a 2 dimentional plane.

In [37]:

```
x= rand.rand(10, 2)
x
```

Out[37]:

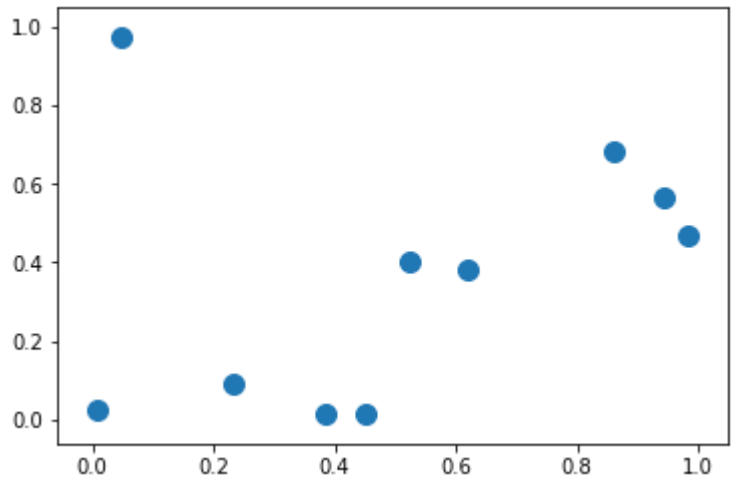
```
array([[0.00706631, 0.02306243],
       [0.52477466, 0.39986097],
       [0.04666566, 0.97375552],
       [0.23277134, 0.09060643],
       [0.61838601, 0.38246199],
       [0.98323089, 0.46676289],
       [0.85994041, 0.68030754],
       [0.45049925, 0.01326496],
       [0.94220176, 0.56328822],
       [0.3854165 , 0.01596625]])
```

In [38]:

```
plt.scatter(x[:,0], x[:,1], s=100)
```

Out[38]:

```
<matplotlib.collections.PathCollection at 0x7f74d1cb9d30>
```



Now we will compute the distance between each pair of points

In [39]:

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

Out[39]:

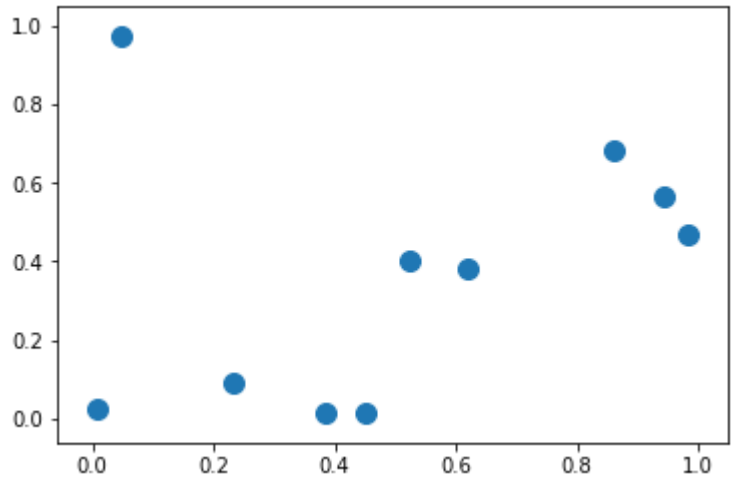
```
array([[0.        , 0.40999909, 0.90538547, 0.05550496, 0.50287983,
        1.14976739, 1.15936537, 0.19672877, 1.16632222, 0.14319923],
       [0.40999909, 0.        , 0.55794316, 0.18090431, 0.00906581,
        0.21465798, 0.19098635, 0.15497331, 0.20095384, 0.16679585],
       [0.90538547, 0.55794316, 0.        , 0.81458763, 0.67649219,
        1.13419594, 0.74752753, 1.08562368, 0.9704683 , 1.03211241],
       [0.05550496, 0.18090431, 0.81458763, 0.        , 0.23387834,
        0.70468321, 0.74108843, 0.05338715, 0.72671958, 0.0288717 ],
       [0.50287983, 0.00906581, 0.67649219, 0.23387834, 0.        ,
        0.14021843, 0.1470605 , 0.16449241, 0.13755476, 0.18859392],
       [1.14976739, 0.21465798, 1.13419594, 0.70468321, 0.14021843,
        0.        , 0.06080186, 0.48946337, 0.01100053, 0.56059965],
       [1.15936537, 0.19098635, 0.74752753, 0.74108843, 0.1470605 ,
        0.06080186, 0.        , 0.61258786, 0.02046045, 0.66652228],
       [0.19672877, 0.15497331, 1.08562368, 0.05338715, 0.16449241,
        0.48946337, 0.61258786, 0.        , 0.54429694, 0.00424306],
       [1.16632222, 0.20095384, 0.9704683 , 0.72671958, 0.13755476,
        0.01100053, 0.02046045, 0.54429694, 0.        , 0.60957115],
       [0.14319923, 0.16679585, 1.03211241, 0.0288717 , 0.18859392,
        0.56059965, 0.66652228, 0.00424306, 0.60957115, 0.        ]])
```

In [40]:

```
plt.scatter(x[:, 0], x[:, 1], s=100)
# draw lines from each point to its two nearest neighbors
K = 2
nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
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')
```

```
---------------------------------------------------------------------------
------
TypeError                                 Traceback (most recent call
 last)
<ipython-input-40-3f993167f0ca> in <module>
      4 nearest_partition = np.argpartition(dist_sq, K + 1, axis=1)
      5 for i in range (x.shape[0]):
----> 6     for j in nearest_partition[i, :k+1]:
      7         # plot a line from X[i] to X[j]
      8         # use some zip magic to make it happen:

TypeError: can only concatenate list (not "int") to list
```



The above cell is supposed to draw connections between the dots. Some how it has refused to do this.

Structured Arrays.Imagine that we have several categories of data on a number of people (say, name, age, and weight). We can create a structured array using a compound data type specification:

In [41]:

```
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),
                          'formats':('U10', 'i4', 'f8')}) #formats':((np.str_, 10),
print(data.dtype)
```

```
[('name', '<U10'), ('age', '<i4'), ('weight', '<f8')]
```

Here 'U10' translates to “Unicode string of maximum length 10,” 'i4' translates to “4-byte (i.e., 32 bit) integer,” and 'f8' translates to “8-byte (i.e., 64 bit) float.”

In [42]:

```
name = ['Alice', 'Bob', 'Cathy', 'Doug']
age = [25, 45, 37, 19]
weight = [55.0, 85.5, 68.0, 61.5]
data['name'] = name
data['age'] = age
data['weight'] = weight
print(data)
```

```
[('Alice', 25, 55. ) ('Bob', 45, 85.5) ('Cathy', 37, 68. )
 ('Doug', 19, 61.5)]
```

In [43]:

```
data[1] #remember array indexes are not quoted.
```

Out[43]:

```
('Bob', 45, 85.5)
```

Using Boolean masking, this even allows you to do some more sophisticated opera- tions such as filtering on age:

In [44]:

```
# Get names where age is under 30
data[data['age'] < 30]['name']
```

Out[44]:

```
array(['Alice', 'Doug'], dtype='<U10')
```

Array Slicing: Accessing Subarrays

In [45]:

```
x = np.arange(20)
print(x[:5]) # first five elements
print(x[4:7]) # middle subarray
print(x[::2]) # Every other element in additions of 2
print(x[3::2]) # Every other element begining with 3 in additions of 2
```

```
[0 1 2 3 4]
[4 5 6]
[ 0  2  4  6  8 10 12 14 16 18]
[ 3  5  7  9 11 13 15 17 19]
```

Multidimensional subarrays

In [46]:

```
print(x2[:2, :3]) # two rows, three columns
print(x2[:3, ::2]) # all rows, every column in additions of two
```

```
[[12  3  1]
 [ 5  2  0]]
[[12  1]
 [ 5  0]
 [ 1  3]]
```

Accessing array rows and columns.

In [47]:

```
print(x2[:, 0]) # first column of x2
print(x2[1,:]) # print second row of x2.
```

```
[12  5  1]
[5 2 0 2]
```

In the case of row access, the empty slice can be omitted for a more compact syntax: print(x2[0]) # equivalent to x2[0, :]

Creating copies of arrays

In [48]:

```
x2_copy = x2[:2, :2].copy() #the copy command is used.
x2_copy
```

Out[48]:

```
array([[12,  3],
       [ 5,  2]])
```

Reshaping of Arrays

The most flexible way of doing this is with the reshape() method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, you can do the following: grid = np.arange(1, 10).reshape((3, 3))

[Note that for this to work, the size of the initial array must match the size of the reshaped array.] Another common reshaping pattern is the conversion of a one-dimensional array into a two-dimensional row or column matrix.

ARRAY CONCATINATION AND SPLITTING

In [49]:

```
# Look at the following example
x = np.array([1, 2, 3])
y = np.array([3, 2, 1])
np.concatenate([x, y])
```

Out[49]:

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

When working with arrays of mixed dimensions, it can be clearer to use the np.vstack(vertical stack; the no of columns has to be the same) and np.hstack (horizontal stack; the no of rows has to be the same) functions. Similarly, np.dstack will stack arrays along the third axis:

In [50]:

```
x = np.array([[1, 2, 3,11,14,56],
              [4,5,6,33,54,12,]])
grid = np.array([[9, 8, 7],
                 [6, 5, 4]])
y = np.hstack([x, grid])
y
```

Out[50]:

```
array([[ 1,  2,  3, 11, 14, 56,  9,  8,  7],
       [ 4,  5,  6, 33, 54, 12,  6,  5,  4]])
```

Splitting of arrays

This is implemented by the functions np.split, np.hsplit, and np.vsplit

In [51]:

```
x = [1, 2, 3, 99, 99, 3, 2, 1]
x1, x2, x3 = np.split(x, [3, 5])
print(x1, x2, x3)
```

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

Notice that N split points lead to N + 1 subarrays.

The related functions np.hsplit and np.vsplit are similar: Similarly, np.dsplit will split arrays along the third axis.

In [52]:

```
grid = np.arange(16).reshape((4, 4))
upper, lower = np.vsplit(grid, [2]) # Splits the array into two. (hsplit splits hor
print(upper)
print(lower)
```

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

Absolute value

Just as NumPy understands Python's built-in arithmetic operators, it also understands Python's built-in absolute value function:

In [59]:

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

Out[59]:

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

In [60]:

```
# This ufunc can also handle complex data, in which the absolute value returns the
x = np.array([3 - 4j, 4 - 3j, 2 + 0j, 0 + 1j])
abs(x)
```

```
1.53 µs ± 5.72 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops
each)
```

Trigonometric functions

NumPy provides a large number of useful ufuncs, and some of the most useful for the data scientist are the trigonometric functions. We'll start by defining an array of angles:

In [62]:

```
theta = np.linspace(0, np.pi, 3)
# Now we can compute some trigonometric functions on these values:
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]
```

In [64]:

```
# Another common type of operation available in a NumPy ufunc are the exponentials:
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))
print("ln(x) =", np.log(x))
print("log2(x) =", np.log2(x))
print("log10(x) =", np.log10(x))
```

```
x = [1, 2, 3]
e^x = [ 2.71828183  7.3890561  20.08553692]
2^x = [2. 4. 8.]
3^x = [ 3  9 27]
ln(x) = [0.         0.69314718 1.09861229]
log2(x) = [0.        1.        1.5849625]
log10(x) = [0.         0.30103    0.47712125]
```

The Reduce Operation

A reduce repeatedly applies a given operation to the elements of an array until only a single result remains. For example, calling reduce on the add ufunc returns the sum of all elements in thet array:

In [65]:

```
x = np.arange(1, 6)
np.add.reduce(x)
```

Out[65]:

```
15
```

In [67]:

```
#np.multiply.reduce(x) # returns a product of all array elements:
np.add.accumulate(x) # returns x together with the final value of addition
```

Out[67]:

```
array([ 1,  3,  6, 10, 15])
```

Summing the Values in an Array

In [68]:

```
L = np.random.random(100)
sum(L) #np.sum(L) returns the same result but it is however faster.
```

Out[68]:

```
48.62061851147834
```

Minimum and Maximum

In [72]:

```
print(min(L))
print(max(L))
#np.min(K) and np.max(K) generate same results though much more faster
```

```
0.005061583846218687
0.9856504541106007
```

Aggregation

For example, we can find the minimum value within each column in a two dimentional array by specifying axis=0:

In [74]:

```
M = np.random.random((3, 4))
print(M.min(axis=0)) #returns minimum value per column(.max returns maximum value)
print(M.min(axis=0)) #returns minimum value per row.(.max returns maximum value)
```

```
[0.29444889 0.3701587  0.01545662 0.31692201]
[0.29444889 0.3701587  0.01545662 0.31692201]
```

Other Usefull aggregation functions

```
        print("25th percentile: ", np.percentile(heights, 25)) #heights is an a
    rray.
        print("Median: ", np.median(heights))
        print("75th percentile: ", np.percentile(heights, 75))
```

ARRAY COMPUTATIONS

Broadcasting: A set of rules for applying binary ufuncs on arrays of different sizes.Example:

In [78]:

```
a = np.array([0, 1, 2])
M = np.ones((3, 3))
M + a
```

Out[78]:

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

Note that while we've been focusing on the + operator here, these broadcasting rules apply to any binary ufunc.