Exploratory Data Analysis- BDS613B

Prepared By,
Dr. Anitha DB
Associate Professor & Head
Department of CSE-Data Science
ATME College of Engineering, Mysuru

Module1 : Introduction to Python and NumPy

- Getting Started in IPython and Jupyter,
- Enhanced Interactive Features,
- The Basics of NumPy Arrays,
- Sorted Arrays,
- Structured Data: NumPy’s Structured Arrays

The Basics of NumPy Arrays

Basic array manipulations here:

- *Attributes of arrays*: Determining the size, shape, memory consumption, and data types of arrays
- *Indexing of arrays*: Getting and setting the value of individual array elements
- *Slicing of arrays*: Getting and setting smaller subarrays within a larger array
- *Reshaping of arrays*: Changing the shape of a given array
- *Joining and splitting of arrays*: Combining multiple arrays into one, and splitting one array into many

NumPy Array Attributes-ndim, shape, size, dtype, itemsize, nbytes

Three random arrays →a one-dimensional, two-dimensional, and three-dimensional array.

NumPy's random number generator.

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

Each array has attributes `ndim` (the number of dimensions), `shape` (the size of each dimension), and `size` (the total size of the array):

```
In[2]: 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
```

Another useful attribute is the dtype, the data type of the array

```
In[3]: print("dtype:", x3.dtype)

dtype: int64
```

Other attributes include itemsize, which lists the size (in bytes) of each array element, and nbytes, which lists the total size (in bytes) of the array

```
In[4]: print("itemsize:", x3.itemsize, "bytes")
       print("nbytes:", x3.nbytes, "bytes")

itemsize: 8 bytes
nbytes: 480 bytes
```

Array Indexing: Accessing Single Elements

1. In a one-dimensional array, we can access the ith value (counting from zero) by specifying the desired index in square brackets.

```
In[5]: x1
Out[5]: array([5, 0, 3, 3, 7, 9])
In[6]: x1[0]
Out[6]: 5
In[7]: x1[4]
Out[7]: 7
```

To index from the end of the array, you can use negative indices:

```
In[8]: x1[-1]
Out[8]: 9
In[9]: x1[-2]
Out[9]: 7
```

Array Indexing: Accessing Single Elements

2. In a multidimensional array, we access items using a comma-separated tuple of indices

```
In[10]: x2

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

In[11]: x2[0, 0]

Out[11]: 3

In[12]: x2[2, 0]

Out[12]: 1

In[13]: x2[2, -1]

Out[13]: 7
```

3. We can also modify values using any of the above index notation

```
In[14]: x2[0, 0] = 12
        x2

Out[14]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])
```

4. NumPy arrays have a fixed type.

If we attempt to insert a floating-point value to an integer array, the value will be silently truncated.

```
In[15]: x1[0] = 3.14159  # this will be truncated!
        x1

Out[15]: array([3, 0, 3, 3, 7, 9])
```

Array Slicing: Accessing Subarrays

Square brackets can also be used to access subarrays with the slice notation, marked by the colon (:) character.

The NumPy slicing syntax to access a slice of an array x is as follows

x[start:stop:step]

If any of these are unspecified, they default to the values start=0, stop=size of dimension, step=1.

Array Slicing: Accessing Subarrays

One-dimensional subarrays

```
In[16]: x = np.arange(10)
        x
Out[16]: array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
In[17]: x[:5]  # first five elements
Out[17]: array([0, 1, 2, 3, 4])
In[18]: x[5:]  # elements after index 5
Out[18]: array([5, 6, 7, 8, 9])
In[19]: x[4:7]  # middle subarray
Out[19]: array([4, 5, 6])
In[20]: x[::2]  # every other element
Out[20]: array([0, 2, 4, 6, 8])
In[21]: x[1::2]  # every other element, starting at index 1
Out[21]: array([1, 3, 5, 7, 9])
```

Array Slicing: Accessing Subarrays

A potentially confusing case is when the step value is negative. In this case, the defaults for start and stop are swapped. This becomes a convenient way to reverse an array:

```
In[22]: x[::-1]  # all elements, reversed

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

In[23]: x[5::-2]  # reversed every other from index 5

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

Array Slicing: Accessing Subarrays

Multidimensional subarrays

Multidimensional slices work in the same way, with multiple slices separated by com mas. For example:

```
In[24]: x2

Out[24]: array([[12,  5,  2,  4],
                [ 7,  6,  8,  8],
                [ 1,  6,  7,  7]])

In[25]: x2[:2, :3]  # two rows, three columns

Out[25]: array([[12,  5,  2],
                [ 7,  6,  8]])

In[26]: x2[:3, ::2]  # all rows, every other column

Out[26]: array([[12,  2],
                [ 7,  8],
                [ 1,  7]])
```

Array Slicing: Accessing Subarrays

Multidimensional subarrays

Finally, subarray dimensions can even be reversed together:

```
In[27]: x2[::-1, ::-1]

Out[27]: array([[ 7,  7,  6,  1],
                [ 8,  8,  6,  7],
                [ 4,  2,  5, 12]])
```

Accessing array rows and columns. One commonly needed routine is accessing single rows or columns of an array. We can do this by combining indexing and slicing, using an empty slice marked by a single colon (:):

```
In[28]: print(x2[:, 0])  # first column of x2

[12  7  1]
```

```
In[29]: print(x2[0, :])  # first row of x2

[12  5  2  4]
```

In the case of row access, the empty slice can be omitted for a more compact syntax:

```
In[30]: print(x2[0])  # equivalent to x2[0, :]

[12  5  2  4]
```

Array Slicing: Accessing Subarrays

Subarrays as no-copy views

Array slices return views rather than copies of the array data.

```
x2 = np.random.randint(10, size=(3, 4))  # Two-dimensional array
```

```
In[31]: print(x2)
```

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

Let's extract a 2×2 subarray from this:

```
In[32]: x2_sub = x2[:2, :2]
        print(x2_sub)
```

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

Now if we modify this subarray, we'll see that the original array is changed! Observe:

```
In[33]: x2_sub[0, 0] = 99
        print(x2_sub)
```

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

```
In[34]: print(x2)
```

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

This default behavior is actually quite useful: it means that when we work with large datasets, we can access and process pieces of these datasets without the need to copy the underlying data buffer.

Array Slicing: Accessing Subarrays

Creating copies of arrays

It is sometimes useful to instead explicitly copy the data within an array or a subarray. This can be most easily done with the copy() method

```
In[35]: x2_sub_copy = x2[:2, :2].copy()
        print(x2_sub_copy)

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

If we now modify this subarray, the original array is not touched:

```
In[36]: x2_sub_copy[0, 0] = 42
        print(x2_sub_copy)

[[42  5]
 [ 7  6]]

In[37]: print(x2)

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

Array Reshaping

Reshaping of Arrays

Another useful type of operation is reshaping of arrays. The most flexible way of doing reshaping is with the reshape() method. For example, if you want to put the numbers 1 through 9 in a 3×3 grid, we can do the following:

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

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

Note : The size of the initial array must match the size of the reshaped array.

Array Reshaping

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

```
In[39]: x = np.array([1, 2, 3])

        # row vector via reshape
        x.reshape((1, 3))

Out[39]: array([[1, 2, 3]])

In[40]: # row vector via newaxis
        x[np.newaxis, :]

Out[40]: array([[1, 2, 3]])
```

```
In[41]: # column vector via reshape
        x.reshape((3, 1))

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

In[42]: # column vector via newaxis
        x[:, np.newaxis]

Out[42]: array([[1],
                [2],
                [3]])
```

Array Concatenation and Splitting

It is possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays

Concatenation of arrays

Concatenation, or joining of two arrays in NumPy, can be performed through the following routines.

- np.concatenate,
- np.vstack,
- np.hstack.

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

```
In[43]: x = np.array([1, 2, 3])
        y = np.array([3, 2, 1])
        np.concatenate([x, y])

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

We can also concatenate more than two arrays at once

```
In[44]: z = [99, 99, 99]
        print(np.concatenate([x, y, z]))

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

Array Concatenation and Splitting

np.concatenate can also be used for two-dimensional arrays:

```
In[45]: grid = np.array([[1, 2, 3],
                         [4, 5, 6]])

In[46]: # concatenate along the first axis
        np.concatenate([grid, grid])

Out[46]: array([[1, 2, 3],
                [4, 5, 6],
                [1, 2, 3],
                [4, 5, 6]])

In[47]: # concatenate along the second axis (zero-indexed)
        np.concatenate([grid, grid], axis=1)

     Out[47]: array([[1, 2, 3, 1, 2, 3],
                     [4, 5, 6, 4, 5, 6]])
```

Array Concatenation and Splitting

For working with arrays of mixed dimensions, it can be clearer to use the **np.vstack** (vertical stack) and **np.hstack** (horizontal stack) functions:

```
In[48]: x = np.array([1, 2, 3])
        grid = np.array([[9, 8, 7],
                         [6, 5, 4]])

        # vertically stack the arrays
        np.vstack([x, grid])

Out[48]: array([[1, 2, 3],
                [9, 8, 7],
                [6, 5, 4]])

In[49]: # horizontally stack the arrays
        y = np.array([[99],
                      [99]])
        np.hstack([grid, y])

Out[49]: array([[ 9,  8,  7, 99],
                [ 6,  5,  4, 99]])
```

Array Concatenation and Splitting

Splitting of arrays

The opposite of concatenation is splitting, which is implemented by the functions np.split, np.hsplit, and np.vsplit. For each of these, we can pass a list of indices giving the split points:

```
In[50]: 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]
```

Array Concatenation and Splitting

Splitting of arrays

The related functions np.hsplit and np.vsplit are similar:

```
In[51]: grid = np.arange(16).reshape((4, 4))
        grid

Out[51]: array([[ 0,  1,  2,  3],
                [ 4,  5,  6,  7],
                [ 8,  9, 10, 11],
                [12, 13, 14, 15]])

In[52]: upper, lower = np.vsplit(grid, [2])
        print(upper)
        print(lower)

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

```
In[53]: left, right = np.hsplit(grid, [2])
        print(left)
        print(right)

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

Similarly, np.dsplit will split arrays along the third axis.

Sorting Arrays

Insertion sorts, Selection sorts, Merge sorts, Quick sorts, Bubble sorts

A simple **selection sort** repeatedly finds the minimum value from a list, and makes swaps until the list is sorted.

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

       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

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

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

Bogosort

```
In[3]: def bogosort(x):
           while np.any(x[:-1] > x[1:]):
               np.random.shuffle(x)
           return x

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

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

Fortunately, Python contains built-in sorting algorithms that are much more efficient than either of the simplistic algorithms just shown.

Sorting Arrays

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

np.sort uses quick sort algorithm. To return a sorted version of the array without modifying the input, we can use np.sort:

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

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

If we prefer to sort the array in-place, we can instead use the sort method of arrays:

```
In[6]: x.sort()
       print(x)

[1 2 3 4 5]
```

A related function is argsort, which instead returns the indices of the sorted elements:

```
In[7]: x = np.array([2, 1, 4, 3, 5])
       i = np.argsort(x)
       print(i)

[1 0 3 2 4]
```

Sorting Arrays

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

The first element of this result gives the index of the smallest element, the second value gives the index of the second smallest, and so on. These indices can then be used (via fancy indexing) to construct the sorted array if desired:

```
In[8]: x[i]

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

Sorting along rows or columns

A useful feature of NumPy's sorting algorithms is the ability to sort along specific rows or columns of a multidimensional array using the axis argument.

```
In[9]: 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]]

In[10]: # sort each column of X
        np.sort(X, axis=0)

Out[10]: 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]])

In[11]: # sort each row of X
        np.sort(X, axis=1)

Out[11]: 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]])
```

Sorting Arrays

Partial Sorts: Partitioning

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:

```
In[12]: x = np.array([7, 2, 3, 1, 6, 5, 4])
        np.partition(x, 3)

Out[12]: array([2, 1, 3, 4, 6, 5, 7])
```

Note that the first three values in the resulting array are the three smallest in the array, and the remaining array positions contain the remaining values. Within the two partitions, the elements have **arbitrary order**.

Sorting Arrays

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

```
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]]
```

```
In[13]: np.partition(X, 2, axis=1)

Out[13]: 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]])
```

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. Finally, just as there is a **np.argsort** that computes indices of the sort, there is a **np.argpartition** that computes indices of the partition.

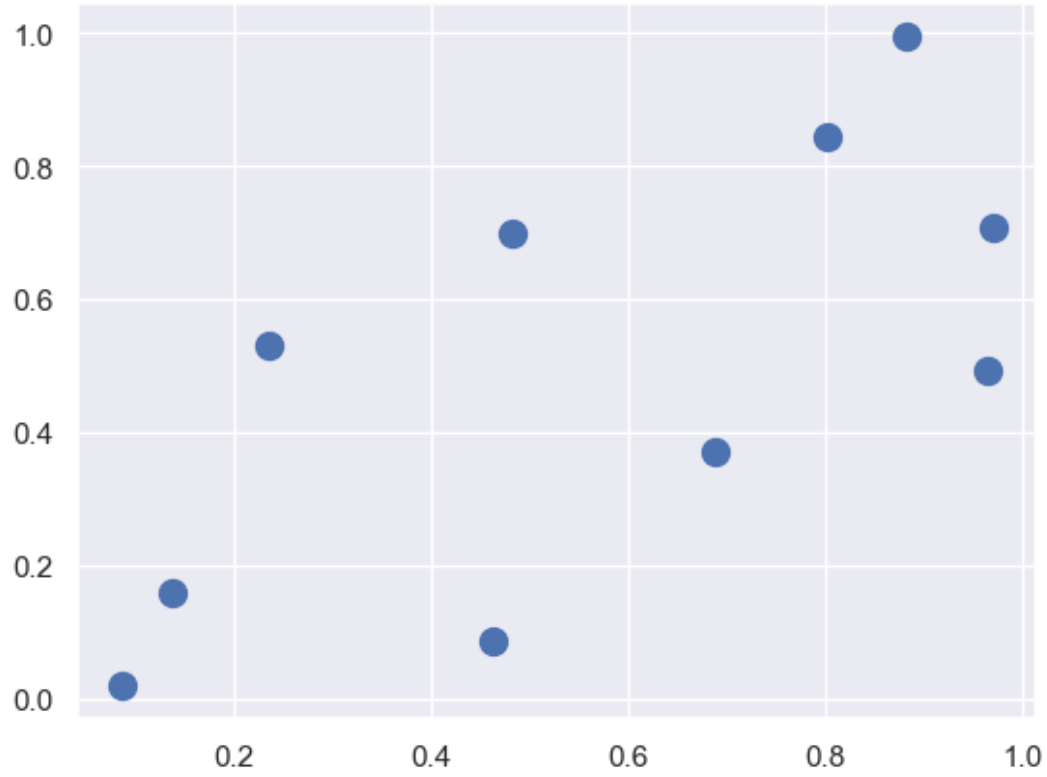
Sorting Arrays

Example: k-Nearest Neighbors (np.argpartition)

The **argsort** function can be used to find the nearest neighbors of each point in a set along multiple.

We'll start by creating a random set of points on a two-dimensional plane. Using the standard convention, we'll arrange these in a 10×2 array:

```
[2]: import numpy as np

[6]: X = np.random.rand(10, 2)

[7]: X

[7]: array([[0.48314126, 0.7000092 ],
            [0.97095671, 0.70858595],
            [0.13776795, 0.15908742],
            [0.88351929, 0.99522761],
            [0.08701521, 0.01938545],
            [0.46313165, 0.08557444],
            [0.68840048, 0.37204209],
            [0.96544289, 0.49249755],
            [0.23607378, 0.53095145],
            [0.80167063, 0.84277537]])

[8]: import matplotlib.pyplot as plt

[9]: import seaborn; seaborn.set() # Plot styling

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



Sorting Arrays

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

```
[13]: differences = X[:, np.newaxis, :] - X[np.newaxis, :, :]
      differences.shape
```

```
[13]: (10, 10, 2)
```

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

```
[14]: (10, 10, 2)
```

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

```
[15]: (10, 10)
```

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

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

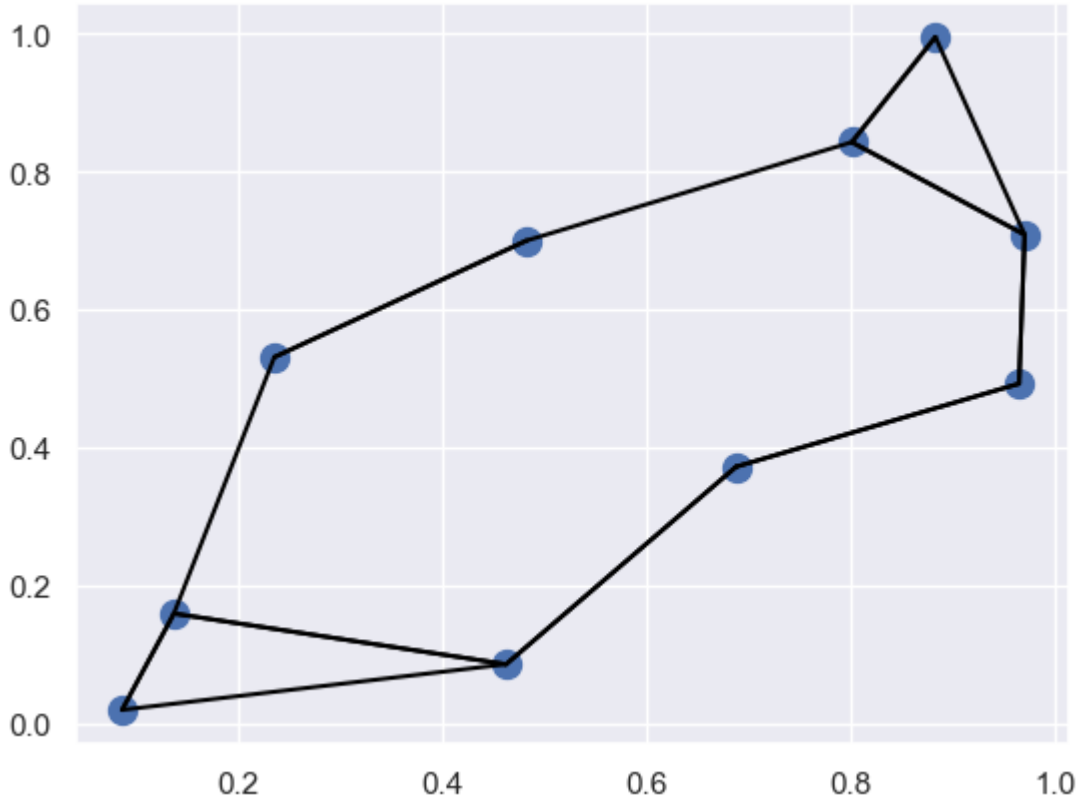
```
[17]: nearest = np.argsort(dist_sq, axis=1)
      print(nearest)
```

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

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

Sorting Arrays

```
[19]: plt.scatter(X[:, 0], X[:, 1], s=100)
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')
```



Each point in the plot has lines drawn to its two nearest neighbors.

Structured Data: NumPy's Structured Arrays

Structured Data: NumPy's Structured Arrays

Imagine that we have several categories of data on a number of people (say, name, age, and weight), and we'd like to store these values for use in a Python program. It would be possible to store these in three separate arrays:

```
#Structured Data: NumPy's Structured Arrays
```

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

But this is a bit clumsy. There's nothing here that tells us that the three arrays are related; it would be more natural if we could use a **single structure** to store all of this data. NumPy can handle this through structured arrays, which are arrays with compound data types.

Structured Data: NumPy's Structured Arrays

```
#Create a structured array using a compound data type specification
```

```
# Use a compound data type for structured arrays
data = np.zeros(4, dtype={'names':('name', 'age', 'weight'),'formats':('U10', 'i4', 'f8')})
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."

```
#Now we can fill the created an empty container array with our lists of values:
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)]
```

Structured Data: NumPy's Structured Arrays

```
# We can refer to values either by index or by name in structured arrays
# Get all names
data['name']
```

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

```
# Get first row of data
data[0]
```

```
('Alice', 25, 55.)
```

```
# Get the name from the last row
data[-1]['name']
```

```
'Doug'
```

Structured Data: NumPy's Structured Arrays

Filtering

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

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

Structured Data: NumPy's Structured Arrays

Creating Structured Arrays : **Structured array data types** can be specified in a number of ways.

```
# Creating structured Array
np.dtype({'names':('name', 'age', 'weight'),'formats':('U10', 'i4', 'f8')})
```

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

```
#For clarity, numerical types can be specified with Python types or NumPy dtypes instead:
np.dtype({'names':('name', 'age', 'weight'),'formats':((np.str_, 10), int, np.float32)})
```

```
dtype([('name', '<U10'), ('age', '<i4'), ('weight', '<f4')])
```

```
#A compound type can also be specified as a list of tuples:
np.dtype([('name', 'S10'), ('age', 'i4'), ('weight', 'f8')])
```

```
dtype([('name', 'S10'), ('age', '<i4'), ('weight', '<f8')])
```

Structured Data: NumPy's Structured Arrays

```
"""#If the names of the types do not matter to you, you can specify the types alone in a
 comma-separated string:"""
np.dtype('S10,i4,f8')
```

```
dtype([('f0', 'S10'), ('f1', '<i4'), ('f2', '<f8')])
```

Table 2-4. NumPy data types

Character	Description	Example
'b'	Byte	np.dtype('b')
'i'	Signed integer	np.dtype('i4') == np.int32
'u'	Unsigned integer	np.dtype('u1') == np.uint8
'f'	Floating point	np.dtype('f8') == np.int64
'c'	Complex floating point	np.dtype('c16') == np.complex128
'S','a'	string	np.dtype('S5')
'U'	Unicode string	np.dtype('U') == np.str_
'V'	Raw data (void)	np.dtype('V') == np.void

Structured Data: NumPy's Structured Arrays

More Advanced Compound Types

```
#create a data type with a mat component consisting of a 3×3 floating-point matrix:
tp = np.dtype([('id', 'i8'), ('mat', 'f8', (3, 3))])
X = np.zeros(1, dtype=tp)
print(X[0])
print(X['mat'][0])
```

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

Each element in the X array consists of an id and a 3×3 matrix.

Structured Data: NumPy's Structured Arrays

RecordArrays: Structured Arrays with a Twist

NumPy also provides the **np.recarray** class, which is almost identical to the structured arrays just described, but with one additional feature: fields can be accessed as attributes rather than as dictionary keys.

```
In[16]: data_rec = data.view(np.recarray)
        data_rec.age

Out[16]: array([25, 45, 37, 19], dtype=int32)
```

The downside is that for record arrays, there is some extra overhead involved in accessing the fields, even when using the same syntax. We can see this here:

```
In[17]: %timeit data['age']
        %timeit data_rec['age']
        %timeit data_rec.age

1000000 loops, best of 3: 241 ns per loop
100000 loops, best of 3: 4.61 µs per loop
100000 loops, best of 3: 7.27 µs per loop
```

THANK YOU