NumPy

NumPy

- Stands for Numerical Python
- Is the fundamental package required for high performance computing and data analysis
- NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.
- It provides
  - ndarray for creating multiple dimensional arrays
  - Standard math functions for fast operations on entire arrays of data without having to write loops
  - Internally stores data in a contiguous block of memory, independent of other built-in Python objects, use much less memory than built-in Python sequences.

NumPy ndarray vs list

- One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python.
- Whenever you see “array,” “NumPy array,” or “ndarray” in the text, with few exceptions they all refer to the same thing: the ndarray object.
- NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

```
import numpy as np
my_arr = np.arange(1000000)
my_list = list(range(1000000))
```

ndarray

- ndarray is used for storage of homogeneous data
  - i.e., all elements the same type
- Every array must have a shape and a dtype
- Supports convenient slicing, indexing and efficient vectorized computation
  - Arrays are important because they enable you to express batch operations on data without writing any for loops. We call this vectorization.

```
data1 = [6, 7.5, 8, 0, 1]
arr1 =  np.array(data1)


#list of lists
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]

arr2 = np.array(data2)
print(arr2.ndim)   #2
print(arr2.shape)  # (2,4)
```

Creating ndarrays

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

array = np.zeros((2,3))
[[0. 0. 0.]
 [0. 0. 0.]]

array = np.ones((2,3))
[[1. 1. 1.]
 [1. 1. 1.]]

array = np.eye(3)
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]

array = np.arange(0, 10, 2)
[0, 2, 4, 6, 8]

array = np.random.randint(0, 10, (3,3))
[[6 4 3]
 [1 5 6]
 [9 8 5]]
```

Arithmetic with NumPy Arrays

- Any arithmetic operations between equal-size arrays applies the operation element-wise:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
[[1. 2. 3.]
 [4. 5. 6.]]

print(arr * arr)
[[ 1.  4.  9.]
 [16. 25. 36.]]

print(arr - arr)
[[0. 0. 0.]
 [0. 0. 0.]]
```

Arithmetic with NumPy Arrays

- Arithmetic operations with scalars propagate the scalar argument to each element in the array:

```
arr = np.array([[1., 2., 3.], [4., 5., 6.]])
print(arr)
[[1. 2. 3.]
 [4. 5. 6.]]
```

```
print(arr **2)
[[ 1.  4.  9.]
 [16. 25. 36.]]
```

- Comparisons between arrays of the same size yield boolean arrays:

```
arr2 = np.array([[0., 4., 1.], [7., 2., 12.]])
print(arr2)
[[ 0.  4.  1.] [ 7.  2. 12.]]

print(arr2 > arr)
[[False  True False]
 [ True False  True]]
```

Indexing and Slicing

- One-dimensional arrays are simple; on the surface they act similarly to Python lists:

```
arr = np.arange(10)
print(arr)          # [0 1 2 3 4 5 6 7 8 9]
print(arr[5])       #5
print(arr[5:8]) #[5 6 7]
arr[5:8] = 12
print(arr)      #[ 0 1 2 3 4 12 12 12 8 9]
```

Indexing and Slicing

- As you can see, if you assign a scalar value to a slice, as in arr[5:8] = 12, the value is propagated (or broadcasted) to the entire selection.
- An important first distinction from Python's built-in lists is that array slices are views on the original array.
  - This means that the data is not copied, and any modifications to the view will be reflected in the source array.

```
arr = np.arange(10)
print(arr)          # [0 1 2 3 4 5 6 7 8 9]
arr_slice = arr[5:8]
print(arr_slice)          # [5 6 7]
arr_slice[1] = 12345
print(arr)          # [    0     1     2     3     4     5 12345     7     8     9]
arr_slice[:] = 64
print(arr)          # [ 0  1  2  3  4 64 64 64  8  9]
```

Indexing

- In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[2])       # [7 8 9]
```

- Thus, individual elements can be accessed recursively. But that is a bit too much work, so you can pass a comma-separated list of indices to select individual elements.
- So these are equivalent:

```
print(arr2d[0][2])     # 3
print(arr2d[0, 2])     #3
```

Slicing

- Consider the two-dimensional array from before, arr2d. Slicing this array is a bit different:

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(arr2d[:2])
[[1 2 3]
 [4 5 6]]
```

- You can pass multiple slices just like you can pass multiple indexes:

```
print(arr2d[:2, 1:])
[[2 3]
 [5 6]]
```

- Of course, assigning to a slice expression assigns to the whole selection:

```
arr2d[:2, 1:] = 0
print(arr2d)
[[1 0 0]
 [4 0 0]
 [7 8 9]]
```

Data Types

- np.int64
- np.float32
- np.complex
- np.bool
- np.object
- np.string_
- np.unicode_

- Signed 64-bit integer types
- Standard double-precision floating point
- Complex numbers represented by 128 floats
- Boolean type storing TRUE and FALSE values
- Python object type
- Fixed-length string type
- Fixed-length unicode type

Aggregation functions

- a.sum()
- a.min()
- b.max(axis=0)
- b.cumsum(axis=1)
- a.mean()
- b.median()
- a.corrcoef()
- np.std(b)

- Array-wise sum
- Array-wise minimum value
- Maximum value of an array row
- Cumulative sum of the elements
- Mean
- Median
- Correlation coefficient
- Standard deviation

Saving & Loading Text Files

```
a= np.genfromtxt("my_file.csv", delimiter=',')
np.savetxt("other_file.csv”, a, delimiter=" ")
```