Python for Machine Learning and Data Analysis

Handan Liu, PhD

h.liu@northeastern.edu

Northeastern University

Spring 2019

Numerical Analysis and Data Exploration with NumPy Arrays

Lecture 3

Preface

- This class and next two classes will outline techniques for effectively loading, storing, and manipulating in-memory data in Python.
- The topic is very broad:
  - datasets can come from a wide range of sources and a wide range of formats, including be:
    - collections of documents
    - collections of images
    - collections of sound clips
    - collections of numerical measurements
    - or nearly anything else.
  - Despite this apparent heterogeneity, it will help us to think of all data fundamentally as **arrays of numbers**.
- For this reason, efficient storage and manipulation of numerical arrays is absolutely fundamental to the process of doing data science.
  - Numpy packages
  - Pandas packages

Contents

- Introduction to Numpy
- Basics of Numpy Arrays
- Numpy Array Attributes
- Array Indexing, Slicing, Reshaping
- Array Concatenating and Splitting
- Aggregations
- Numpy UFuncs
- Broadcasting

Introduction to Numpy

- What is Numpy?
  - Core library for scientific computing in Python. It provides a high-performance multidimensional array object, and tools for working with these arrays.
  - Short for Numerical Python, provides an efficient interface to store and operate on dense data buffers.
  - Like Python’s built-in list type, but Numpy arrays provide much more efficient storage and data operations as the arrays grow larger in size.
- NumPy arrays form the core of nearly the entire ecosystem of data science tools in Python, so time spent learning to use NumPy effectively will be valuable no matter what aspect of data science interests you

Basics of Numpy Arrays

- Data manipulation in Python is nearly synonymous with NumPy array manipulation: even newer tools like Pandas (next classes) are built around the NumPy array.
- We'll cover a few categories of 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

- Each array has attributes:
  - ndim: the number of dimensions
  - shape: the size of each dimension
  - size: the total size of the array
- Other useful attributes:
  - dtype: the data type of the array (e.g. integer, floating point)
  - itemsize: lists the size (in bytes) of each array element
  - nbytes: lists the total size (in bytes) of the array

  -- In general, we expect that nbytes is equal to itemsize times size.

Array Indexing: Accessing Single Elements

- Indexing in Numpy is quite similiar with Python's standard list indexing.
- In a one-dimensional array, the i^{th} value (counting from zero) can be accessed by specifying the desired index in square brackets, just as with Python lists.
- In a multi-dimensional array, items can be accessed using a comma-separated tuple of indices.
- Keep in mind that, unlike Python lists, NumPy arrays have a fixed type.

Array Slicing: Accessing Subarrays (1)

- Also use square brackets to access subarrays with the slice notation, marked by the colon (:) character.
- The NumPy slicing syntax follows that of the standard Python list; to access a slice of an array x, use this:
  - 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 (2)

- Accessing sub-arrays in:
  - One-dimensional 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:
  - Multi-dimensional subarrays
    - Multi-dimensional slices work in the same way, with multiple slices separated by commas.
  - Accessing array rows and columns
    - combine indexing and slicing, using an empty slice marked by a single colon (:)
  - Subarrays as no-copy views
    - 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!!
  - Creating copies of arrays: copy()

Reshaping of Arrays

- Doing this is with the *reshape* method

```
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:
  - Or using the *newaxis* keyword within a slice operation.

Array Concatenation and Splitting

- Concatenation of arrays:
  - or joining of two arrays in NumPy, is primarily accomplished using the routines: np.concatenate, np.vstack, and np.hstack.
  - np.concatenate takes a tuple or list of arrays as its first argument.
  - For working with arrays of mixed dimensions, it can be clearer to use the np.vstack (vertical stack) and np.hstack (horizontal stack) functions.
- 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.
  - Notice that N split-points, leads to N + 1 subarrays.
  - The related functions np.hsplit and np.vsplit are similar.

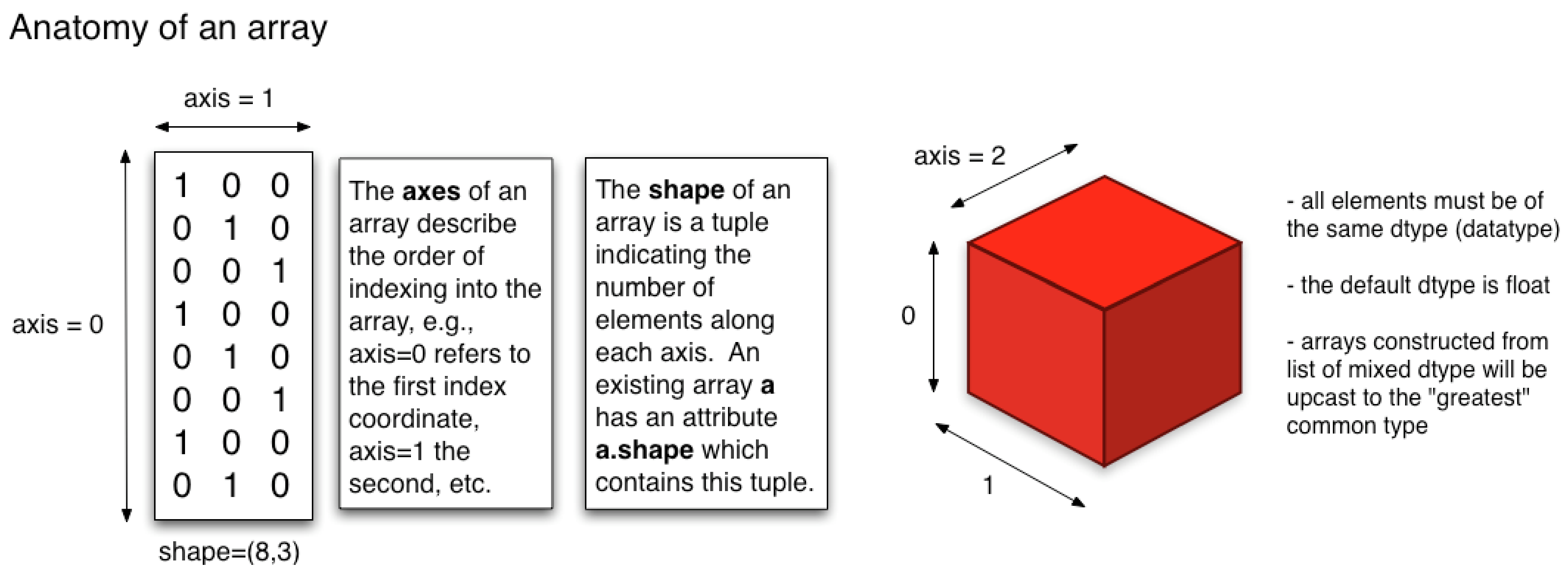
Aggregations: Min, Max, and Everything In Between

- Common summary statistics: mean and standard deviation to summarize the “typical” values in dataset.
- Other aggregations are useful as well:
  - sum, product, median, minimum and maximum, quantiles, etc.
- Summing the Values in an Array
  - Numpy.sum()
  - Compare with Python build-in function sum()
    - The syntax of Python sum function is quite similar to that of NumPy's sum function, and the result is the same in the simplest case.
    - Numpy.sum is computed much more quickly as it executes the operation in compiled code.
    - the sum function and the np.sum function are not identical:
      - optional arguments have different meanings;
      - np.sum is aware of multiple array dimensions

Aggregations (2)

- Minimum and Maximum
  - Numpy.min() and numpy.max() compared with Python build-in function min(), max()
    - NumPy's corresponding functions have similar syntax, and again operate much more quickly.
- Multi dimensional aggregates
  - One common type of aggregation operation is an aggregate along a row or column.
  - Aggregation functions take an additional argument specifying the axis along which the aggregate is computed. For example,
    - find the minimum value within each column by specifying axis=0:

PS. Axis: To understand the axis intuitively, refer the picture below



$$\begin{array}{ccc}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\\ 1 & 0 & 0\\ 0 & 1 & 0\end{array}$$

The **axes** of an array describe the order of indexing into the array, e.g., axis=0 refers to the first index coordinate, axis=1 the second, etc.

The **shape** of an array is a tuple indicating the number of elements along each axis. An existing array **a** has an attribute **a.shape** which contains this tuple.

- all elements must be of the same dtype (datatype)

- the default dtype is float

- arrays constructed from list of mixed dtype will be upcast to the "greatest" common type

The shape of the array in the above figure is shape=(8, 3). ndarray.shape will return a tuple where the entries correspond to the length of the particular dimension. Here, 8 corresponds to length of axis 0 whereas 3 corresponds to length of axis 1.

Aggregations (3)

- Other aggregation functions

Function Name	Description	Function Name	Description
np.sum	Compute sum of elements	np.argmin	Find index of minimum value
np.prod	Compute product of elements	np.argmax	Find index of maximum value
np.mean	Compute mean of elements	np.median	Compute median of elements
np.std	Compute standard deviation	np.percentile	Compute rank-based statistics of elements
np.var	Compute variance	np.any	Evaluate whether any elements are true
np.min	Find minimum value	np.all	Evaluate whether all elements are true
np.max	Find maximum value		

Example 1:
What is the Average Height of US Presidents?

- Aggregates available in NumPy can be extremely useful for summarizing a set of values.
- As a simple example, let's consider the heights of all US presidents.
- This data is available in the file L03_heights.csv, which is a simple comma-separated list of labels and values.
- Pandas package to read the file and extract the information (note that the heights are measured in centimeters).
- To compute a variety of summary statistics: mean height, standard deviation, minimum height, maximum height, median height, etc.
- These aggregates are some of the fundamental pieces of exploratory data analysis that you should explore in more depth in the future.

Numpy Arrays: Universal Functions

- The Slowness of Loops:
  - Python's default implementation (known as CPython) does some operations very slowly:
    - Like PyPy, Cython, Numba, each of these has its strengths and weaknesses, but it is safe to say that none of the three approaches has yet surpassed the reach and popularity of the standard CPython engine.
  - The sluggishness of Python: many small operations are being repeated - for instance looping over arrays to operate on each element.

NumPy Arrays: Universal Functions (2)

- A *vectorized* operation in Numpy: *universal functions* (*ufuncs*)
- The main purpose of *ufuncs* is to quickly execute repeated operations on values in NumPy arrays.
  - Make repeated calculations on array elements much more efficient.
  - Many of the most common and useful arithmetic *ufuncs* available in the NumPy package.
  - Extremely flexible – operation between a scalar and an array, and between two arrays.
  - Act on multi-dimensional arrays as well.
- Note:
  - Computations using vectorization through ufuncs are nearly always more efficient than their counterpart implemented using Python loops, especially as the arrays grow in size.
  - Any time you see such a loop in a Python script, you should consider whether it can be replaced with a vectorized expression.

Exploring NumPy's UFuncs

- *Ufuncs* exist in two flavors:
  - *unary ufuncs*: operate on a single input, and
  - *binary ufuncs*: operate on two inputs.
- Array arithmetic:
  - The standard addition, subtraction, multiplication, and division
  - Also a unary ufunc for negation, and a ** operator for exponentiation, and a % operator for modulus.
  - These can be strung together however you wish, and the standard order of operations is respected.
  - Each of these arithmetic operations are simply convenient wrappers around specific functions built into NumPy; for example, the + operator is a wrapper for the add function.

The following table lists the arithmetic operators implemented in NumPy:

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

Exploring NumPy's Ufuncs (3)

- Absolute value
  - The corresponding NumPy ufunc is np.absolute, (alias) np.abs.
- Trigonometric functions
  - np.sin(), np.cos(), np.tan(), etc.
- Exponents and logarithms
  - np.exp(), np.exp2(), np.power(); np.log(), np.log2(), np.log10()
  - np.expm1(), np.log1p() -- useful for precision with very small input.
- Other many ufuncs:
  - hyperbolic trig functions, bitwise arithmetic, comparison operators, conversions from radians to degrees, rounding and remainders, and much more. -- See Numpy documentation
- Specialized ufuncs: a submodule *scipy.special* -- compute specialized and obscure mathematical function on your data

Exploring NumPy's Ufuncs (4) – Advanced Ufunc Features

- Specifying output
  - Store data to memory directly: out
- Aggregates
  - For binary ufuncs, there are some interesting aggregates that can be computed directly from the object.
- Outer products
  - Any ufunc can compute the output of all pairs of two different inputs using the outer method. This allows you, in one line, to do things like create a multiplication table.

Exploring NumPy's Ufuncs (4) – Broadingcasting

- Broadcasting is simply a set of rules for applying binary ufuncs (e.g., addition, subtraction, multiplication, etc.) on arrays of different sizes.
- See examples.
- The advantage of NumPy's broadcasting is that this duplication of values does not actually take place, but it is a useful mental model as we think about broadcasting.

Rules of Broadcasting

- Broadcasting in NumPy follows a strict set of rules to determine the interaction between the two arrays:
  - 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.
- To make these rules clear, let's consider a few examples in detail.

Broadcasting example 1

- Let's look at adding a two-dimensional array to a one-dimensional array:

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

- Let's consider an operation on these two arrays. The shape of the arrays are:

```
M.shape = (2, 3)
a.shape = (3, )
```

- We see by **rule 1** that the array a has fewer dimensions, so we pad it on the left with ones:

```
M.shape -> (2, 3)
a.shape -> (1, 3)
```

- By **rule 2**, we now see that the first dimension disagrees, so we stretch this dimension to match:

```
M.shape -> (2, 3)
a.shape -> (2, 3)
```

- The shapes match, and we see that the final shape will be (2, 3):

```
M + a
array([[ 1.,  2.,  3.],
       [ 1.,  2.,  3.]])
```

Broadcasting example 2

- Let's take a look at an example where both arrays need to be broadcast:

```
a = np.arange(3).reshape((3, 1))
b = np.arange(3)
```

- Again, we'll start by writing out the shape of the arrays:

```
a.shape = (3, 1)
b.shape = (3,)
```

- Rule 1 says we must pad the shape of b with ones:

```
a.shape -> (3, 1)
b.shape -> (1, 3)
```

- And rule 2 tells us that we upgrade each of these ones to match the corresponding size of the other array:

```
a.shape -> (3, 3)
b.shape -> (3, 3)
```

- Because the result matches, these shapes are compatible. We can see this here:

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

Broadcasting example 3

- Now let's take a look at an example in which the two arrays are not compatible:

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

- This is just a slightly different situation than in the first example: the matrix M is transposed. How does this affect the calculation? The shape of the arrays are:

```
M.shape = (3, 2)
a.shape = (3, )
```

- Again, rule 1 tells us that we must pad the shape of a with ones:

```
M.shape -> (3, 2)
a.shape -> (1, 3)
```

- By rule 2, the first dimension of a is stretched to match that of M:

```
M.shape -> (3, 2)
a.shape -> (3, 3)
```

- Now we hit rule 3 – the final shapes do not match, so these two arrays are incompatible, as we can observe by attempting this operation:

```
M + a
------
ValueError…………
```

- Note the potential confusion here:
  - you could imagine making a and M compatible by, say, padding a's shape with ones on the right rather than the left.
  - But this is not how the broadcasting rules work!
  - That sort of flexibility might be useful in some cases, but it would lead to potential areas of ambiguity. If right-side padding is what you'd like, you can do this explicitly by reshaping the array.

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

Broadcasting in Practice

- Centering an array
  - Imagine you have an array of 10 observations, each of which consists of 3 values. Store this in a 3 x 10 array.
  - Compute the mean of each feature using the mean aggregate across the first dimension;
  - Center the X array by subtracting the mean (this is a broadcasting operation);
  - To double-check that we've done this correctly, we can check that the centered array has near zero mean:

- Plotting a two-dimensional function
  - One place that broadcasting is very useful is in displaying images based on two-dimensional functions. If we want to define a function $z=f(x,y)$, broadcasting can be used to compute the function across the grid.
  - Use Matplotlib to plot this two-dimensional array.

Some concepts:

- Python list vs. Numpy array
  - A list is the Python equivalent of an array, but is resizeable and can contain elements of different types.
  - The main benefits of using NumPy arrays should be smaller memory consumption and better runtime behavior.
- np.array vs. np.ndarray
  - np.array is just a convenience function to create an ndarray, it is not a class itself.
  - You can also create an array using np.ndarray, but it is not the recommended way. From the docstring of np.ndarray.

The End!

Assignment: Practice the example code I ran in this class.