Re(Introduction) to NumPy

Aidan Hunt, University of Washington

Learning Objectives

After this lesson, students will be able to:

- Identify the key differences between Lists and NumPy array
- Interpret the attributes and methods of NumPy arrays
- Use NumPy functions (such as numpy.array, numpy.zeroes) to create NumPy arrays
- Using indexing, slicing, and logical indexing to extract data from NumPy arrays

---

Check-in / Announcements

Homework 3 posted Homework 2 grading underway

Where we are at:

- Finished with "intro" material
- Tools we've discussed largely apply in any programming language
- Lectures going forward are assuming you are experts in functions, loops, built-in data structures

Framing

An important skill in programming is choosing the correct data type for a particular application.

We have the following built-in data structures

- List
- Dictionary
- Tuple (immutable list)
- Set (like a list with unique, unordered components)

But we're engineers, we work with big datasets, timeseries, etc. We work with matrices!

- **How might we create a matrix from the built-in data types we have? (brainstorm)**
    - Nested lists
    - Outer lists represent rows, inner lists represent columns
    - Use for loops for elementwise operations, multiplying lists together, etc.

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

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

- **What are some challenges to this approach?**

- For loops are slow in Python
- Need to keep lists compatible
  - Lists can grow/shrink
  - Lists can have any data type
- Additional complexity for variety of matrix operations

Key Takeaway: lots of overhead

Solution: dedicated datatype for matrices

The NumPy package

Importing the numpy package

1. Verify that NumPy is installed on your machine (e.g, `conda list` in Anaconda prompt)
2. Import the package: `import numpy as np`
   - `np` is shorthand that we will use when interfacing with the package
   - Only need to import once at the beginning of a script

What are we getting with NumPy?

1. A new data structure: NumPy arrays (vectors, matrices, higher-order arrays)
2. A whole bunch of functions for creating, reshaping, altering these arrays
3. A whole bunch of functions for doing math with these arrays

Creating a basic array

We can create a basic array by providing our nested lists to `numpy.array` :

In [3]:

```
# Import the package
import numpy as np

# Create array using list of lists
# Each element of the first list is a row
exArray = np.array([ [1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12] ])
print(exArray)
```

```
[[ 1  2  3  4]
 [ 5  6  7  8]
 [ 9 10 11 12]]
```

NumPy Arrays vs Lists

Aside from the convenience of representing matrices, how else do NumPy arrays vary from Lists?

Contents

- Lists can contain any combination of anything -> flexibility is expensive
- NumPy arrays are homogenous (only one data type) -> streamlines operations

In [4]:
```python
# Using a string in the construction of the array makes everything strings
np.array([[1, 2, 'test'], [4, 5, 6], [7, 8, 9]])
```

Out[4]:
```
array([['1', '2', 'test'],
       ['4', '5', '6'],
       ['7', '8', '9']], dtype='<U11')
```

In [5]:
```python
# Mixing types causes data type of "object".
# This is essentially a list,
np.array([[1, 2, 'test'], [4, {'key', 'value'}, 6], [7, 8, 9]])
```

Out[5]:
```
array([[1, 2, 'test'],
       [4, {'key', 'value'}, 6],
       [7, 8, 9]], dtype=object)
```

Size

- Lists can grow/shrink dynamically
- NumPy arrays are fixed size at creation (memory efficient) -> preallocation
- Arrays are generalized to n-dimensions

Access

- Access similarly!
- Indexing/slicing generalized to higher dimensions

In [6]:
```python
# Get element in second row, third column
exArray[1,2]
```

Out[6]:
```
7
```

In [7]:
```python
# Get all elements in last row
exArray[-1,:]
```

Out[7]:
```
array([ 9, 10, 11, 12])
```

In [8]:
```python
# Get elements from first row (inclusive) to third row (exclusive), and from second colu
exArray[:2,1:]
```

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

Math

- With lists, must use a for loop for element-wise calculations (lists are not vectors)
- NumPy arrays are built for element-wise ("vectorized") calculations

In [9]:
```python
# Multiply two lists together using for loop (via list comprehension)
list1 = [5, 10, 15]
list2 = [3, 6, 9]
list3 = [num1 * num2 for num1, num2 in zip(list1, list2)]

print(list3)
```

```
[15, 60, 135]
```

In [10]:
```python
# Multiplying two arrays together elementwise is much easier
array1 = np.array(list1)
```

```
array2 = np.array(list2)
array3 = array1 * array2

print(array3)
```

```
[ 15  60 135]
```

Speed

- NumPy arrays are faster than lists
  - Homogenous data types
  - Many operations use compiled C code (fast)

Reference Semantics

- Lists are a collection of references to data in memory
- A NumPy array is a single reference to a "densely packed" collection of data in memory

---

More ways to create arrays

Most often, however, we will use useful NumPy array creation routines.

`numpy.zeros` : get array of zeros of the specified size

In [11]:
```
# Create a 10 x 5 matrix of zeroes
np.zeros([10,5])
```

Out[11]:
```
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
```

`numpy.ones` : get array of ones of the specified size. Use multiplication to make an array of any number!

In [12]:
```
# Create a 5x5 matrix of ones
np.ones([5, 5])
```

Out[12]:
```
array([[1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1.]])
```

In [13]:
```
# Create a length 10 vector where each element is pi
np.ones(10) * np.pi
```

Out[13]:
```
array([3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265,
       3.14159265, 3.14159265, 3.14159265, 3.14159265, 3.14159265])
```

`numpy.linspace` : Get vector of `N` evenly spaced points from `start` (inclusive) to `end` (also inclusive).

In [14]:
```
# Create 500 points spaced evenly between 0 and 12345
```

```
start = 0
end = 12345
N = 25
np.linspace(start, end, N)
```

Out[14]:
```
array([    0.   ,   514.375,  1028.75 ,  1543.125,  2057.5  ,  2571.875,
        3086.25 ,  3600.625,  4115.   ,  4629.375,  5143.75 ,  5658.125,
        6172.5  ,  6686.875,  7201.25 ,  7715.625,  8230.   ,  8744.375,
        9258.75 ,  9773.125, 10287.5  , 10801.875, 11316.25 , 11830.625,
       12345.   ])
```

`numpy.arange` : Get vector of points from `start` (inclusive) to `end` (exclusive) spaced `dx` apart. (This is essentially the inverse of `numpy.linspace()`, where you specify the number of points between `start` and `end`, and the spacing is automatically calculated. Here, you specify the spacing between points, and the number of points is between `start` and `end` is automatically calculated).

In [15]:
```
# Generate points between -pi and pi, separated by pi/12
start = -np.pi
end = np.pi
dx = np.pi / 12
np.arange(start, end+dx, dx) # End is exclusive!
```

Out[15]:
```
array([-3.14159265e+00, -2.87979327e+00, -2.61799388e+00, -2.35619449e+00,
       -2.09439510e+00, -1.83259571e+00, -1.57079633e+00, -1.30899694e+00,
       -1.04719755e+00, -7.85398163e-01, -5.23598776e-01, -2.61799388e-01,
        1.77635684e-15,  2.61799388e-01,  5.23598776e-01,  7.85398163e-01,
        1.04719755e+00,  1.30899694e+00,  1.57079633e+00,  1.83259571e+00,
        2.09439510e+00,  2.35619449e+00,  2.61799388e+00,  2.87979327e+00,
        3.14159265e+00])
```

`numpy.random.rand` : generate a matrix of random numbers between 0 and 1 of the specified size

In [16]:
```
# Generate a 3x3 matrix of random numbers
np.random.rand(3,3)
```

Out[16]:
```
array([[0.87913997, 0.27934358, 0.95637495],
       [0.13063966, 0.19756426, 0.02805115],
       [0.50153532, 0.8953277 , 0.54801648]])
```

In [17]:
```
# Create a 3x3 matrix with 5's on the diagonal
np.eye(3) * 5
```

Out[17]:
```
array([[5., 0., 0.],
       [0., 5., 0.],
       [0., 0., 5.]])
```

Array Attributes and Methods

Attributes

Array have attributes that describe things about them. (**Show ndarray attribute documentaion**)

Access these with "dot" notation: `arrayName.attributeName`

In [41]:
```
# Shape of an array
exArray.shape
```

Out[41]:
```
2
```

In [34]:
```
# Data type of an array
exArray.dtype
```

Out[34]:
```
dtype('int32')
```

Methods

Arrays also have methods, which are functions that belong to the array and can do things with the data in the array.

- These are just like list methods and string methods for lists and strings, respectively.

We also access these with the "dot" notation: `arrayName.methodName(parameters)`

In [42]:
```
# Get the sum of all array elements
exArray.sum()
```

Out[42]:
```
78
```

Many array methods accept an optional parameter `axis`, which specifies the dimension along which to perform an operation.

- `axis=0`: perform operation along rows
- `axis=1`: columns operation along columns

Note: "Along" can be a bit misleading. When we say we perform an operation "along" rows, we don't sum the elements in each individual row. Rather, we sum by adding the i^{th} elements of each row to each other.

In [46]:
```
exArray # Remind ourselves what our example array looks like
```

Out[46]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

In [47]:
```
exArray.sum(axis=0)  # sum elements "along" rows (e.g, summing entries in each column)
```

Out[47]:
```
array([15, 18, 21, 24])
```

In [48]:
```
exArray.sum(axis=1)  # sum elements "along" columns (e.g, summing entries in each row)
```

Out[48]:
```
array([10, 26, 42])
```

Other NumPy functions

In addition to methods that belong to arrays, there are also several other NumPy functions for general use (called "routines" in the documentation):

- Array creation routines (saw these before)
- Array manipulation routines (joining, sorting, reshaping)
    - `concatenate()`
    - `hstack()`
    - `vstack()`
- Generalized math functions

- `sin()`, `cos()`, `exp()`, etc.
- `mean()`, `std()`, other stats functions
- Reading and writing data
- Specific math functions
  - polynomial fitting
  - fft
- And more! There is something for everyone
- Some overlap with array methods
  - e.g., `mean()`

Additional details of arrays

Logical statements and indexing

We can use logical statements to find all array elements that meet a specific criteria. These statements are performed elementwise and returned as an array of `bools`. For example:

In [25]:
```
# Create simple array
exArray = np.array([[2, 4, 6], [4, 8, 12], [6, 12, 18]])
print(exArray)
```

```
[[ 2  4  6]
 [ 4  8 12]
 [ 6 12 18]]
```

In [26]:
```
# Find all elements that are greater than 10
exArray > 10
```

Out[26]:
```
array([[False, False, False],
       [False, False,  True],
       [False,  True,  True]])
```

- Comparison operators (`>`, `>=`, `<`, `<=`, `==`, `!=`) are the same as for lists.
- Logical operators are different for arrays, though!
  - `and` becomes `&`
  - `or` becomes `|`
  - `not` becomes `!`
- AND we **need to be careful of our parentheses**, now, because precedence is a bit different for these operators.

In [30]:
```
# Find all elements that are equal to 4 or 12
(exArray == 4) | (exArray == 12)
```

Out[30]:
```
array([[False,  True, False],
       [ True, False,  True],
       [False,  True, False]])
```

We can also use these indices to perform **logical indexing**. That is, we can extract elements that meet a particular criteria.

In [31]:
```
# Find all elements that are equal to 4 or 12
result = (exArray == 4) | (exArray == 12)
```

```
# Extract these elements
exArray[result]
```

Out[31]:
```
array([ 4,  4, 12, 12])
```

In [32]:
```
# Placing the condition directly in the square brackets:
exArray[exArray != 8]
```

Out[32]:
```
array([ 2,  4,  6,  4, 12,  6, 12, 18])
```

Reference Semantics

Like lists, two variables can point to the same array, and altering the array data will alter both variables:

In [43]:
```
# Create two arrays that share data
arr1 = np.array([1, 2, 3, 4, 5, 6])
arr2 = arr1

# Print contents
print(arr1)
print(arr2)

# Change one of the variables
arr1[0] = 5000

# Show that both variables update
print(arr1)
print(arr2)

# Show that they share the same identity
arr1 is arr2
```

```
[1 2 3 4 5 6]
[1 2 3 4 5 6]
[5000    2    3    4    5    6]
[5000    2    3    4    5    6]
```

Out[43]:
```
True
```

Slicing an array returns a **view** of an array: another look at the same core data:

In [54]:
```
# Slice arr1
someNums = arr1[2:]
print(arr1)
print(someNums)

# Check if this result shares the same core data as arr1
someNums.base is arr1
```

```
[ 5000     2 -5555     8    10    12]
[-5555     8    10    12]
```

Out[54]:
```
True
```

Changing the sliced values changes the core data!

In [55]:
```
# Changing the slice changes the core data
someNums[0] = -5555

print(arr1)
print(someNums)
```

```
[ 5000     2 -5555     8    10    12]
[-5555     8    10    12]
```

However, generally assignments based on calculations made using an array produce a brand new array that is disconnected from the original:

In [56]:
```python
arr1 = np.array([1, 2, 3]) # Create an array
arr2 = arr1 * 5            # Create another array based on a calculation performed with

# Show that arr1 and arr2 are different objects
print(arr1)
print(arr2)
arr1 is arr2
```

```
[1 2 3]
[ 5 10 15]
```

Out[56]:
```
False
```

Why is it like this?

- Memory efficient, especially when working with large datasets.

Bottom line:

- Be mindful of this if working with large data sets.
- If you need a copy, use the `.copy()` method to make one (just like lists).
- Use the array attribute `.base` to check if two arrays share the same core data.
- Look at documentation to see if certain functions return views or copies.