Working with numpy arrays

This video will explore the capabilities of numpy arrays.

Numpy arrays

- Require numpy module to be imported…

```
import numpy
```

- Create from rasters
- Create from a list…

```
>>> lst = [1,2,3,4]
>>> array = numpy.array(lst)
>>> array
array([1,2,3,4])
```

2

The **numpy** module is installed automatically by ArcGIS. The module needs to be imported when working with numpy arrays.

Arcpy's RasterToNumPyArray method converts a raster to an array format.

An array can also be created from a list.

Array statistics

- Statistics available if array contains numbers

```
>>> array = numpy.array([6, 2, 5, 4, 1])
```

- Max value...

```
>>> array.max()
6
```

- Min value...

```
>>> array.min()
1
```

- Mean value...

```
>>> array.mean()
3.6
```

- Standard deviation...

```
>>> array.std()
1.85
```

- Variance...

```
>>> array.var()
3.42
```

3

Arrays can provide basic statistics on the data they contain.

These statistics include the maximum value, the minimum value, the mean, the standard deviation, and the variance.

Array operations

```
>>> array = numpy.array([6, 2, 5, 4, 1])
```

- product of all items…

```
>>> array.prod()
240
```

- cumulative product…

```
>>> array.cumprod()
array([6, 12, 60, 240, 240])
```

- sum of all items…

```
>>> array.sum()
17
```

- cumulative sum…

```
>>> array.cumsum()
array([6, 8, 13, 17, 18])
```

4

Other methods of arrays include:

1) calculating the product of all items, 2) the cumulative product of the items, 3) the sum of all items, and 4) the cumulative sum of the items.

Array math operations

```
>>> array = numpy.array([6, 2, 5, 4, 1])
```

- Arrays can be used with all standard math operators, e.g.

```
>>> array ** 3
array([36, 4, 25, 16, 1])
```

```
>>> array * 3
array([18, 6, 15, 12, 3])
```

- Operation is applied to all items in array

5

Arrays work with all standard math operators. The math operation is applied to each value in the array.

Array math operations

- Multiple arrays can be used in math expressions…

```
>>> array1 = numpy.array([5, 4, 2])
>>> array2 = numpy.array([3, 4, 2])
>>> array 1 * array2
array([15, 16, 4])
```

- Each in item in the 1st array is paired with item in same position of 2nd array.
  - arrays must have compatible sizes
    - e.g. a 2x2 array cannot be in math expression with a 1x3 array

6

Arrays can be added, subtracted, multiplied, or divided.

When multiple arrays are used with a map operator, the data from the arrays are paired by their location in the array. Arrays must have compatible sizes in order to be used in math expresssions.

Processing speed – arrays vs. lists

```
array = numpy.random.randn(10000000)

List = array.tolist()
```

← convert array to list

List approach 4.4sec

```
newLst = []
for val in List:
    val = val*3
    newLst.append(val)
```

List comprehension 2.2sec

```
newLst = [val*3 for val in List]
```

Array approach 0.07sec

```
newArray = array * 3
```

- Arrays are much faster at math operations than lists or list comprehensions.

7

This slide will compare processing speed of using an array with a math operator to equivalent methods using lists. The initial statement creates an array with 10 million random numbers.

The array is converted to a list using the **tolist** method. For the tests, we'll triple each value in the list or array.

For the first list approach, we'll use the list in a standard for loop to multiple each item by 3. This approach took 4.4 seconds.

The second list approach used a list comprehension – the speed was twice as fast as the for loop method.

The array approach was by far the fastest – 60 times faster than the 1st method and 30 times faster than the list comprehension.

Processing speed – arrays vs. lists

```
array = numpy.random.randn(100000)
List = array.tolist()
```

<u>List approach</u> 0.3sec

```
cnt = 0
for val in List:
    if val % 2 == 0:
        cnt += 1
```

<u>Array approach</u> 5.0sec

```
cnt = 0
for val in array:
    if val % 2 == 0:
        cnt += 1
```

- Arrays are much slower than lists when it comes to iterating in a loop.

This slide will compare the processing speed of a list and an array in a loop.

Iterating through the list took 0.3 seconds.

Iterating through an equivalent array took 16 times as long as for the list.

Arrays are much faster at math operations but lists are much faster when you need to iterate through the data collection.

Transpose and dot product

```
>>> array
array([[3, 9]
       [4, 8]
       [5, 7]])
```

- transpose (columns become rows; rows become columns)…

```
>>> array.transpose()
array([3, 4, 5], [9, 8, 7]])
```

```
>>> array1
array([3, 3])
>>> array2
array([4, 8])
```

- dot product of 2 vectors…

```
>>> array1.dot(array2)
36
```

9

Arrays can be transposed which switches the rows and columns.

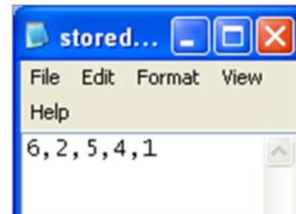
The dot product of two arrays can be calculated.

Storing arrays in text files

- Arrays can be stored in text files…
  - frees up memory used by an array

```
>>> array = numpy.array([6, 2, 5, 4, 1])
>>> File = r“C:\Temp\stored_array.txt”
>>> array.tofile(File, sep = “,”)
```

character to separate values



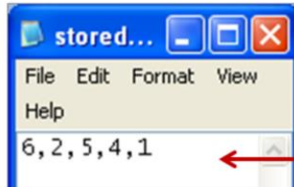
10

The data in an array can be conveniently stored in a text file – this can be useful for temporary data storage when the memory in the array needs to be freed up. On the next slide, we’ll see an array method that can turn the text file back to an array.

The **tofile** method converts the array to a text file. All data in the text file are stored on a single line.

Converting text files to arrays

- Text files can be converted back to an array…



all data should be on one line

```
>>> File = r"C:\Temp\stored_array.txt"

>>> numpy.fromfile(File, dtype = int, sep = ",")
```

data type for array

character separating values

11

Text files can be converted to an array.

All values in the text file should be on a single line.

The **fromfile** method converts the text file to an array. This method along with the **tofile** method allows data to be easily transferred back and forth between internal memory and hard drive storage.

Creating sequential arrays

- Create sequential array – equivalent to the list's range() function. Values can be decimal numbers.

```
>>> numpy.arange(start = 0, stop = 3, step = 0.5)
array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5])
```

increment (can be decimal number)

- Create sequential array with a specified number of values…

include the endpoint?

```
>>> numpy.linspace(start = 0, stop = 3, num = 5, endpoint = True,
retstep = True)
(array([ 0.  ,  0.75,  1.5 ,  2.25,  3.  ]), 0.75)
```

return increment? → retstep

number of values

increment

12

The array has methods similar to the range function for creating sequential values. Unlike the range or xrange functions, the array functions can create decimal numbers.

The **arange** function is equivalent to the **range** function except that it can create decimal numbers.

The **linspace** method creates a specific number of values between the start and stop value.

Creating array

- Create a new array filled with ones…

```
numpy.ones(shape = (2, 3), dtype = numpy.float)
```

rows — columns — or numpy.int

- Convert a given list/array into an array of ones…

```
numpy.ones_like(dataLst, dtype = numpy.float)
```

list/array

- Create a new array filled with zeros…

```
numpy.zeros(shape = (2, 3), dtype = numpy.float)
```

- Convert a given array into an array of zeros…

```
numpy.zeros_like(dataLst, dtype = numpy.float)
```

13

Numpy has a number of methods for creating new arrays. Numpy's **ones** method creates an array in which all values are 1.

The **ones_like** method uses the shape of an existing list or array to create a new array in which all values are 1.

The **zeros** and **zeros_like** methods are equivalent to the previous methods except that they create arrays filled with zeros.

Array modification

- Change the shape of an array without changing data

  numpy.reshape(dataLst, newshape = (2, 3))

  shape of new array

- Extract the unique elements of an array…

  numpy.unique(dataLst)

  >>> numpy.unique([1,2,2,2,3])
  array([1,2,3])

- Trim zeros from 1-D array

  numpy.trim_zeros(dataLst, trim = "fb")

  1-D list/array

  trim front ('f') and/or back ('b')

14

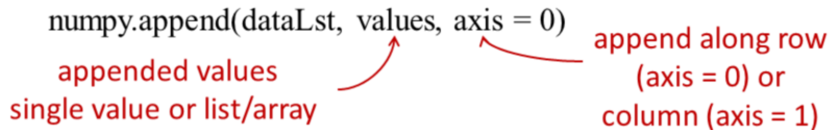
The **reshape** method changes the shape (i.e. number of rows and columns) of an array without changing the data.

The **unique** method creates an new array that contains only the unique values from the input array. This is equivalent to using the **set** function on a list.

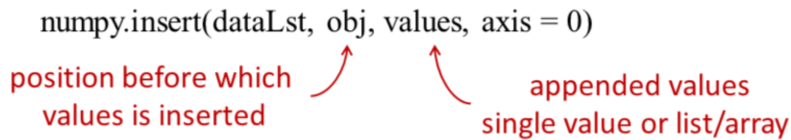
The **trim_zeros** method will remove all zero values from the front or back end of a 1 dimensional array. A 1-dimensional array contains only a single row.

Array modification

- Append values to the end of an array



- Insert values before the given indices



- If axis is not given, dataLst and values will be flattened.

15

Values can be appended to an existing array using the **append** method. Note that a single value or a list or array of values can be appended. If an axis value of zero is specified, then the value will be appended to the end of the row; an axis value of 1 will append the value to the bottom of a column.

The **insert** method allows values to be inserted into any location in the array.

Note that for either the append or insert methods, the resulting array will be flattened if the axis is not specified.

Array indexing

- Return an array satisfying specified condition…

```
>>> Array = numpy.arange(5)
>>> Array[Array>2]
array([3, 4])
```

condition

- Apply operation to a selection of items…

```
>>> Array = numpy.arange(5)
>>> numpy.select(condlist = [Array<=1, Array>=3],
                 choicelist = [Array, Array*2])
array([0, 1, 0, 6, 8])
```

condition

operation applied to items that satisfy the condition

16

Conditional expressions can be used to retrieve only the values from an array that satisfy the expression.

The **select** method allows expressions to be selectively applied to items in the array. The **condlist** parameter specifies one or more conditional expressions. The **choicelist** parameter contains one operation that corresponds to each expression in the **condlist** parameter. In this example, values in the input array that are <= 1 remain unchanged in the output array while values >=3 are squared.