Numpy Module - Arrays, 2D Arrays

numpy Module

numpy is a Python module short for Numerical Python used for scientific computing and data analysis.

numpy provides:

- Multi-dimensional arrays (data structures) for fast, efficient data storage and manipulation.
- Useful functions to manipulate arrays of data.
- Tools for importing and exporting data to and from programs.

numpy Package

numpy functionality used for data analysis includes:

- Array operations for data manipulation.
- Common algorithms such as sorting, unique and set operations.
- Descriptive statistics, aggregating and summarizing data.
- Merging and joining data sets.
- Expressing conditional logic as array expressions instead of loops.

Using numpy Functionality

To use the numpy package, it must first be imported as before.

- `import numpy as np` -> when we want to use the functionality in the package, we will do so using the syntax: `np.functionName(…)`

numpy Data Structure: ndarray

Stores multiple values together in a data structure.

It is a fixed-sized **array** in memory that contains **data** of the same type.

Unlike lists, all elements must be of the same type, an array cannot contain mixed types.

The size of a numpy is fixed when it is created, an array cannot be made larger or smaller; appending/inserting/deleting elements is not permitted.

Numpy arrays are mutable, the values stored in the array can be changed.

Have built in functionality to allow element by element operations, without using loops.

numpy Array Attributes

A Numpy array is a row/grid of values, all of the ***same type***, and is indexed by a tuple of non-negative integers.

Each array has the following attributes:

- `dtype` - the type of data in the array
- `ndim` - the number of dimensions,
- `shape` - the size of each dimension, and
- `size` - the total number of elements in the array

Functions to Create ndarrays

There are several functions that are used to create numpy arrays, these include:

- array(): creates an array with the given sequence of values.
- arange(): creates an array with a specified range of values.
- linspace(): creates an array with n linearly spaced values.
- empty(): creates an array of a given shape with empty elememts.
- zeros(): creates an array with a given shape filled with zeros.
- ones(): creates an array with a given shape filled with ones.
- rand() /randint(): creates an array with a given shape filled with randomly generated values.

Creating numpy Arrays

We can initialize numpy arrays from Python lists, and access elements using square brackets.

To create an array from a list, we can use the array() command, and pass the list as a parameter.

Example – create a one-dimensional array:

```
import numpy as np
z = np.array([4,7,6,9,2])


z
Out[13]: array([4, 7, 6, 9, 2])

print(z)
[4 7 6 9 2]
```

Examples with Attributes

```
r = np.array([87,33,5,16,9,17])

r
Out[29]: array([87, 33,  5, 16,  9, 17])

r.dtype
Out[30]: dtype('int32')

r.ndim
Out[31]: 1

r.shape
Out[32]: (6,)

r.size
Out[33]: 6
```

Creating numpy Arrays - arange

Numpy arrays can also be generated using the `arange` function.

The arange function takes the starting, ending and step values and generates an array containing the given values.

Example:

```
b = np.arange(1,10,2)
b
Out[67]: array([1, 3, 5, 7, 9])
t = np.arange(1,10)
t
Out[65]: array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

linspace()

Function creates evenly spaced float values.

Takes start, end and number of values as parameters.

Generates the given number of values between the specified interval.

```
In [31]: np.linspace?
         np.linspace(-50, 50, 11)

Out[31]: array([-50., -40., -30., -20., -10.,   0.,  10.,  20.,  30.,  40.,  50.])

In [33]: np.linspace(-10, 10)

Out[33]: array([-10.    ,  -9.5918,  -9.1837,  -8.7755,  -8.3673,  -7.9592,
                -7.551 ,  -7.1429,  -6.7347,  -6.3265,  -5.9184,  -5.5102,
                -5.102 ,  -4.6939,  -4.2857,  -3.8776,  -3.4694,  -3.0612,
                -2.6531,  -2.2449,  -1.8367,  -1.4286,  -1.0204,  -0.6122,
                -0.2041,   0.2041,   0.6122,   1.0204,   1.4286,   1.8367,
                 2.2449,   2.6531,   3.0612,   3.4694,   3.8776,   4.2857,
                 4.6939,   5.102 ,   5.5102,   5.9184,   6.3265,   6.7347,
                 7.1429,   7.551 ,   7.9592,   8.3673,   8.7755,   9.1837,
                 9.5918,  10.    ])
```

Creating Special Arrays

We have already used the array and arange functions to create arrays with initial values.

There are also special functions we can use to create arrays with other default values (ones, zeros).

Function	Output
`array`	Creates a numpy array object
`arange`	Creates a numpy array with values within the specified range (start,stop,step)
`ones`	Creates a numpy array containing all ones, with the given shape.
`zeros`	Creates a numpy array containing all zeros, with the given shape.

Creating 1-D arrays: zeros(), ones()

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

In [2]: np.zeros(10)
Out[2]: array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0.])

In [4]: np.ones(10)
Out[4]: array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

Generating Random Arrays

In numpy there is a sub-module, random, that contains functions for generating random arrays.

For example:

- `rand( n )` – generates an array of n random values between 0-1.
- `randint( )` – generates an array of n random integer values between low (inclusive) and high (exclusive).

```
import numpy as np
r_vals = np.random.rand(10)
print(r_vals)
r_vals = np.random.randint(1,26,12)
print(r_vals)
```

Accessing and Updating numpy Arrays

Accessing elements:

```
z[0] -> 4
z[3] -> 9
```

Updating elements:

```
z[2] = 5
z
Out[19]: array([4, 7, 5, 9, 2])
```

Slicing numpy Arrays

Like with lists, we can access parts of a numpy array using the `start:stop:step` syntax.

Reminder – the stop value is up to, but not inclusive of the element specified.

If any of the values are omitted, the default values will be start: 0, stop: end, step: 1.

Examples:

```
z
Out[22]: array([4, 7, 5, 9, 2])
z[1:4:2]
Out[21]: array([7, 9])
z[1:4]
Out[23]: array([7, 5, 9])
z[1::2]
Out[24]: array([7, 9])
z[:4]
Out[25]: array([4, 7, 5, 9])
```

Creating Two-Dimensional Arrays

Two dimensional numpy arrays are just arrays of arrays.

We created the array using the array command and passing a list as a parameter, now we can pass a list of lists.

Example:

```
m = np.array([[3,5,2,4],[7,6,8,8],[1,6,7,7]])
m
Out[35]:
array([ [3, 5, 2, 4],
        [7, 6, 8, 8],
        [1, 6, 7, 7]])
```

Creating 2-D arrays: zeros(), ones()

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

In [3]: np.zeros((3,8))
Out[3]:
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.]])


In [5]: np.ones((3,8))
Out[5]:
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.]])

 In [6]: np.ones((2,3,2))
 Out[6]:
 array([[[1., 1.],
         [1., 1.],
         [1., 1.]],

        [[1., 1.],
         [1., 1.],
         [1., 1.]]])
```

Accessing Elements: Two D Arrays

Syntax: `arrName[ row, col]`

Examples:

```
#accessing elements
m[1,2]
Out[43]: 9
#updating elements
m[2,1] = 12
m
Out[45]:
array([[ 3,  5,  2,  4],
       [ 7,  6,  9,  8],
       [ 1, 12,  0,  7]])
```

Accessing Elements - Slicing

We can also slice 2D arrays, using the `start:stop:step` syntax for each dimension (row, col) separated by a comma.

Examples:

```
m[ 1:2,1:2]
Out[46]: array([[6]])
m[-1,-1]
Out[48]: 7
y = m[2, :]
y
Out[50]: array([ 1, 12,  0,  7])
t = m[2, ]
t
Out[52]: array([ 1, 12,  0,  7])
w = m[2:0:-2,1:3]
w
Out[56]: array([[12,  0]])
```

Element-Wise Operations

Unlike lists, arrays allow us to perform element by element operations without using a loop.

When we apply arithmetic operations to arrays, the operation will be applied to each element of the array.

See: `10_OneDelementOperations.py`

See: `10_TwoDelementOperations.py`

Python Arrays – Mathematical Functions

Basic mathematical functions operate element wise on arrays and are available both as operator overloads and as functions in the numpy module.

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)

Basic mathematical functions operate element wise on arrays and are available both as operator overloads and as functions in the numpy module.

See: 10_numpyArrayMath.py

Numpy Arrays – built in functions

Numpy arrays have many built in functions for computing statistics on an array.
See: `10_numpyArrayFunctions.py`

Function Name	Description
`np.sum()`	Computes the sum of values in given array.
`np.mean()`	Computes the mean(average) value in given array.
`np.max()`	Computes the maximum value in given array.
`np.min()`	Computes the minimum value in given array.
`np.argmax()`	Computes the index of the maximum value in given array.
`np.argmin()`	Computes the index of the minimum value in given array.

Numpy Arrays – Reshaping and Transpose

Another useful type of operation is reshaping arrays. This can be done using the `reshape()` function.
Example: reshape `x = np.arange(1,10)` into a 3 x 3 array:

```
In [4]: M = x.reshape((3, 3))
        M

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

You can also do common matrix operations such as finding the transpose. The transpose of a matrix switches the row and column indices to produce another matrix.
Example: compute the transpose using .T

```
In [5]: M.T

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

numpy Arrays – Matrix Operations

numpy knows how to efficiently do typical matrix operations
Examples:

a matrix-vector product using `np.dot`:

```
In [6]: np.dot(M, [5, 6, 7])
Out[6]: array([ 38,  92, 146])
```

and even more sophisticated operations like eigenvalue decomposition:

```
In [7]: np.linalg.eigvals(M)
Out[7]: array([  1.61168440e+01,  -1.11684397e+00,  -1.3
        0367773e-15])
```

numpy Arrays – Concatenation

It's also possible to combine multiple arrays into one, and to conversely split a single array into multiple arrays.

Concatenation, or joining of two arrays in NumPy, is primarily accomplished through the routines `np.concatenate`, `np.vstack`, and `np.hstack`.

`np.concatenate()` - takes a tuple or list of arrays as its first argument, returns a new array.

```
import numpy as np

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

Output:

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

numpy Arrays – Concatenation

`np.concatenate()` – can be used to concatenate more than two arrays or two-dimensional arrays as well.

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

Output:

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

```
grid = np.array([[1, 2, 3],[4, 5, 6]])
print(np.concatenate([grid, grid])) #default axis = 0
```

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

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

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

numpy – hstack, vstack

The `concatenate` function can be used when the arrays you are joining have the same number of dimensions (one or two).

If you want to join arrays with different dimensions, you should use the functions `vstack` or `hstack`.

`vstack`: stack two or more arrays vertically and returns a new array

`hstack`: stack two or more arrays horizontally and returns a new array.

See: `10_concatenate_vstack_hstack.py`

numpy Arrays – Splitting

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.

```
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]
```

```
grid = np.arange(16).reshape((4, 4))
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]]
```

```
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]]
```

Sorting Arrays

Like lists, numpy arrays can be sorted in place using the `sort()` function.

```
In [67]: arr = np.random.rand(8)

In [68]: arr
Out[68]:
array([0.41857204, 0.52282439, 0.85841995, 0.97523084, 0.52313993,
       0.70206584, 0.37557734, 0.29295251])

In [69]: arr.sort()

In [70]: arr
Out[70]:
array([0.29295251, 0.37557734, 0.41857204, 0.52282439, 0.52313993,
       0.70206584, 0.85841995, 0.97523084])
```

Unique and Set Logic

For one dimensional arrays, numpy has some basic set operations.

The most used is the `unique()` function, which returns the sorted unique values in an array.

```
In [72]: names = np.array(['Bob','Joe','Will','Bob','Will','Joe','Joe'])

In [73]: np.unique(names)
Out[73]: array(['Bob', 'Joe', 'Will'], dtype='<U4')

In [74]: ints = np.array([3,3,3,2,2,1,1,4,4])

In [75]: np.unique(ints)
Out[75]: array([1, 2, 3, 4])
```

Boolean Arrays

A Boolean array is an array that contains True/False values.
A Boolean array may be generated in numpy by using element-wise relational expressions with arrays.

```
In [87]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [88]: average_rainfall > 10
Out[88]: array([ True,  True,  True, False, False,  True,  True])

In [89]: (average_rainfall > 10) & (average_rainfall < 20)
Out[89]: array([False, False,  True, False, False, False,  True])
```

Python assumes that True values are 1 and False values are 0. If we `sum()` the elements of a Boolean array, the True values will be counted.

```
In [106]: np.sum(average_rainfall > 10)
Out[106]: 5
```

Boolean indexing and sub arrays

We can use Boolean arrays to select a subset of an array.

For example if we want to find the rainfall for all days of the week in which it was greater than 10, we can do so using logical indexing.

```
In [87]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [88]: average_rainfall > 10
Out[88]: array([ True,  True,  True, False, False,  True,  True])

In [90]: over_10 = average_rainfall > 10

In [91]: average_rainfall[over_10]
Out[91]: array([55. , 71.3, 10.5, 30.2, 18.8])
```

where() function – array indexes

In the previous example we used the logical array to select the values/elements that met the given condition(s).

If we want to find the indexes of the elements that meet the given condition(s), we can do so using the where function.

The example below shows that the resulting values are the indexes of the elements in the array that meet the given condition. I.e. elements 0,1,2, etc have values over 10.

```
In [96]: average_rainfall = np.array([55.0,71.3,10.5,0.0,8.6,30.2,18.8])

In [97]: np.where(average_rainfall > 10)
Out[97]: (array([0, 1, 2, 5, 6], dtype=int64),)
```

Subarrays and functions

```
import numpy as np
x = np.array([22,41,58,33,17,32])
#returns array of bool values for each element in x where condition is true
r1 = x > 40
print(r1)

#returns array containing elements in x where condition is true
r2 = x[x > 40]
print(r2)

#returns array containing indices of elements in x where condition is true
r3 = np.where(x > 40)
print(r3)
```

Reading data from files with numpy

numpy contains functions that allow us to read text files into numpy arrays.

`loadtxt(filename)`

Reads data from a file (in the current directory unless otherwise specified) into a numpy array.

Default delimiter is whitespace.

Default type is float.

We can add additional parameters when reading the file. For example, if the first row in the file contains headers, we would use the skiprows parameter to start at the second row.

```
loadtxt('myfile.txt',skiprows=1)
```

To read string data from the file use the dtype parameter:

```
loadtxt('myfile.txt',dtype='str')
```

See loadtxt documentation for a full list of parameters.

Writing Data to Files with numpy

Numpy contains functions that allow us to write numpy arrays to text files.

```
savetxt(filename, arrayName)
```

- Writes the data in the numpy array to a text file.
- By default, values in each row are space delimited, and rows are delimited by newlines.

We can add additional parameters when writing data to a file.

See savetxt documentation for a full list of parameters.

Example

The file `10_world_pop_area.txt` contains the population and area for a list of countries. The first row in the file contains headings.

The file `10_countries.txt` contains the text country names.

Write a program that does the following:

- reads the population and areas from the file into an array.
- Read the country names into a separate array.
- Increases the population (in place) by 10%
- Calculates the population density for each country (population per km^2) and stores in a new array.
- Writes the countries and densities to a new file, densities.txt.
- Finds and displays the name of the country with the maximum density, together with the maximum density.

See: `10_numpyFiles.py`

Terms of Use

- This presentation was adapted from lecture materials provided in MIT Introduction to Computer Science and Programming in Python.
- Licenced under terms of Creative Commons License.