UNIQUE System Skills LLC
Our goal is your success

Introduction to Numpy

BY... SAKEEB SHEIKH

UNIQUE System Skills LLC
Our goal is your success

Introduction to Numpy

NumPy is nothing but provides all the libraries to deal with the linear algebra.

- NumPy, which stands for Numerical Python
- NumPy is the foundational package for mathematical computing
- Mathematical and logical operations on arrays
- Operations related to linear algebra. NumPy has inbuilt functions for linear algebra and random generation
- ndarray is the core object in NumPy

Basics ndarray

- Multidimensional array
- Homogeneous collection of values
- Fast and efficient
- Support for mathematical functions
- Primary container for data exchange between python algorithms

Important attributes of an ndarray object are

- **Ndarray.ndim:** the number of axes (dimensions) of the array, the number of dimensions is referred to as rank
- **Ndarray.shape:** the dimensions of the array. This is tuple of integers indicating the size of the array in each dimension. For a matrix within rows and m columns, the shape will be (n,m)
- **Ndarray.size:** the total number of elements of the array.this is equal to the product of the elements of shape
- **Ndarray.dtype:** an object describing the type of the elements in the array

Numpy Arrays

- Numpy arrays are great alternatives to Python Lists. Some of the key advantages of Numpy arrays are that they are fast, easy to work with, and give users the opportunity to perform calculations across entire arrays.

Example

```
# Create 2 new lists height and weight
 height = [1.87,  1.87, 1.82, 1.91, 1.90, 1.85]
 weight = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]
# Import the numpy package as np
 import numpy as np
# Create 2 numpy arrays from height and weight
 np_height = np.array(height)
 np_weight = np.array(weight)
```

Print out the type of np_height

- print(type(np_height))

Datatype and Precedence

- Str
- Complex
- Float
- Int
- Bool

Highest Precedence to Lowest Precedence

Special Functions

- np.arange()
- np.zeros()
- np.ones()
- np.eye()
- np.empty()
- np.copy()
- **np.reshape()**

Element-wise calculations

CODE :-

```
# Calculate bmi
bmi = np_weight / np_height ** 2
# Print the result
print(bmi)
```

Subsetting

- Another great feature of Numpy arrays is the ability to subset. For instance, if you wanted to know which observations in our BMI array are above 23, we could quickly subset it to find out.

```
# For a boolean response
        bmi > 23
# Print only those observations above 23
        bmi[bmi > 23]
```

Exercise

convert the list of weights from a list to a Numpy array. Then, convert all of the weights from kilograms to pounds. Use the scalar conversion of 2.2 lbs per kilogram to make your conversion. Lastly, print the resulting array of weights in pounds.

weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]

Solution

```
weight_kg = [81.65, 97.52, 95.25, 92.98, 86.18, 88.45]
import numpy as np
# Create a numpy array np_weight_kg from weight_kg
np_weight_kg = np.array(weight_kg)
# Create np_weight_lbs from np_weight_kg
np_weight_lbs = np_weight_kg * 2.2
# Print out np_weight_lbs
print(np_weight_lbs)
```

Array Operations
Elementwise operations

- **Basic operations**

  With scalars:

  ```
  >>> a = np.array([1, 2, 3, 4])
  >>> a + 1
  array([2, 3, 4, 5])
  >>> 2**a
  array([ 2, 4, 8, 16])
  ```

All arithmetic operates elementwise:

- **Basic operations**
  - With scalars:

```
>>> a = np.array([1, 2, 3, 4])
>>> a + 1
array([2, 3, 4, 5])
>>> 2**a
array([ 2, 4, 8, 16])
```

```
>>> b = np.ones(4) + 1
>>> a - b
array([-1., 0., 1., 2.])
>>> a * b
array([ 2., 4., 6., 8.])
>>> j = np.arange(5)
>>> 2**(j + 1) - j
array([ 2, 3, 6, 13, 28])
```

These operations are of course much faster than if you did them in pure python:

```
>>> a = np.arange(10000)
>>> %timeit a + 1
10000 loops, best of 3: 24.3 us per loop
>>> l = range(10000)
>>> %timeit [i+1 for i in l]
1000 loops, best of 3: 861 us per loop
```

Array multiplication is not matrix multiplication:

```
>>> c = np.ones((3, 3))
>>> c * c # NOT matrix multiplication!
array([[ 1., 1., 1.],
       [ 1., 1., 1.],
       [ 1., 1., 1.]])
```

Matrix multiplication:

```
>>> c.dot(c)
array([[ 3., 3., 3.],
       [ 3., 3., 3.],
       [ 3., 3., 3.]])
```

Other operations

- **Comparisons:**

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> a == b
array([False, True, False, True], dtype=bool)
>>> a > b
array([False, False, True, False], dtype=bool)
```

Array-wise comparisons:

```
>>> a = np.array([1, 2, 3, 4])
>>> b = np.array([4, 2, 2, 4])
>>> c = np.array([1, 2, 3, 4])
>>> np.array_equal(a, b)
False
>>> np.array_equal(a, c)
True
```

Logical operations:

```
>>> a = np.array([1, 1, 0, 0], dtype=bool)
>>> b = np.array([1, 0, 1, 0], dtype=bool)
>>> np.logical_or(a, b)
array([ True, True, True, False], dtype=bool)
>>> np.logical_and(a, b)
array([ True, False, False, False], dtype=bool)
```

Transcendental functions:

```
>>> a = np.arange(5)
>>> np.sin(a)
array([ 0. , 0.84147098, 0.90929743, 0.14112001, -0.7568025 ])
>>> np.log(a)
array([ -inf, 0. , 0.69314718, 1.09861229, 1.38629436])
>>> np.exp(a)
array([ 1. , 2.71828183, 7.3890561 , 20.08553692, 54.59815003])
```

Transposition:

```
>>> a = np.triu(np.ones((3, 3)), 1) # see help(np.triu)
>>> a
array([[ 0., 1., 1.],
       [ 0., 0., 1.],
       [ 0., 0., 0.]])
>>> a.T
array([[ 0., 0., 0.],
       [ 1., 0., 0.],
       [ 1., 1., 0.]])
```

- **The transposition is a view**
- As a results, the following code **is wrong** and will **not make a matrix symmetric**:

```
>>> a += a.T
```

It will work for small arrays (because of buffering) but fail for large one, in unpredictable ways.

NOTE :-

- *Note*
- **Linear algebra**
- The sub-module numpy.linalg implements basic linear algebra, such as solving linear systems, singular value decomposition, etc.

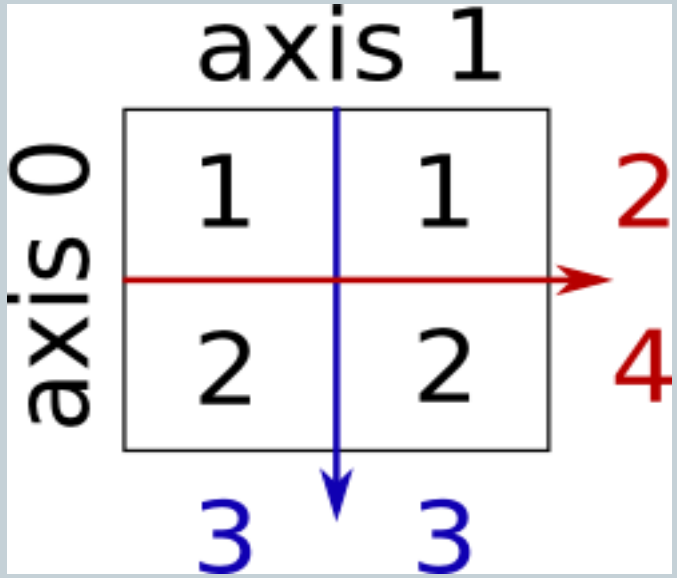
Basic reductions

- **Computing sums**

```
>>> x = np.array([1, 2, 3, 4])
>>> np.sum(x)
10
>>> x.sum()
10
```

Sum by rows and by columns:

```
>>> x = np.array([[1, 1], [2, 2]])
>>> x
array([[1, 1],
       [2, 2]])
>>> x.sum(axis=0) # columns (first dimension)
array([3, 3])
>>> x[:, 0].sum(), x[:, 1].sum()
(3, 3)
>>> x.sum(axis=1) # rows (second dimension)
array([2, 4])
>>> x[0, :].sum(), x[1, :].sum()
(2, 4)
```



Same idea in higher dimensions:

```
>>> x = np.random.rand(2, 2, 2)
>>> x.sum(axis=2)[0, 1]
1.14764...
>>> x[0, 1, :].sum()
1.14764...
```

Other reductions

```
>>> x = np.array([1, 3, 2])
>>> x.min()
1
>>> x.max()
3
>>> x.argmin() # index of minimum
0
>>> x.argmax() # index of maximum
1
```

Logical operations:

```
>>> np.all([True, True, False])
False
>>> np.any([True, True, False])
True
```

Can be used for array comparisons:

```
>>> a = np.zeros((100, 100))
>>> np.any(a != 0)
False
>>> np.all(a == a)
True
>>> a = np.array([1, 2, 3, 2])
>>> b = np.array([2, 2, 3, 2])
>>> c = np.array([6, 4, 4, 5])
>>> ((a <= b) & (b <= c)).all()
True
```

Statistics:

```
>>> x = np.array([1, 2, 3, 1])
>>> y = np.array([[1, 2, 3], [5, 6, 1]])
>>> x.mean()
1.75
>>> np.median(x)
1.5
>>> np.median(y, axis=-1) # last axis
array([ 2., 5.])
>>> x.std() # full population standard dev.
0.82915619758884995
```

Example

- cat data/populations.txt

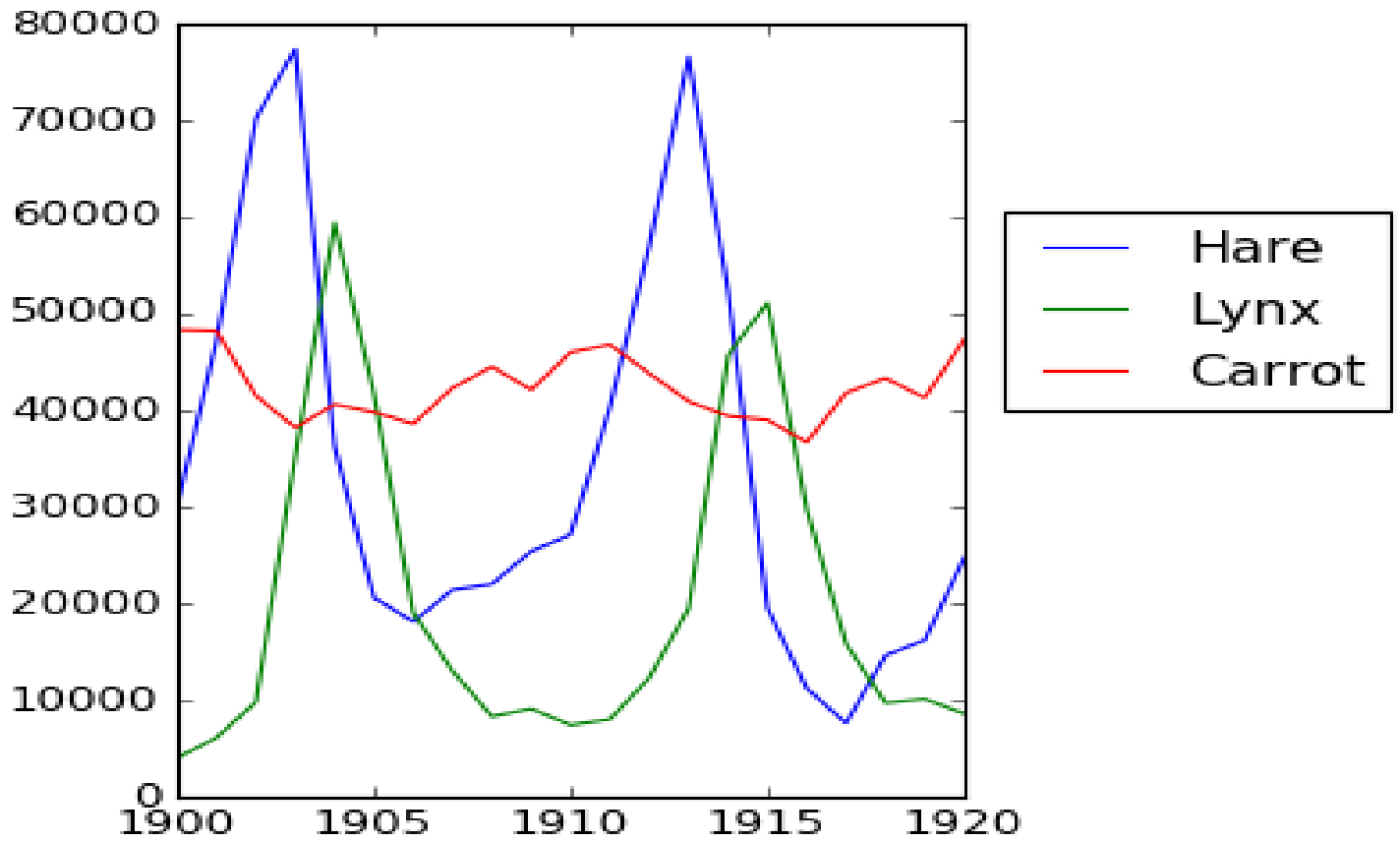
First, load the data into a NumPy array:

```
>>> data = np.loadtxt('data/populations.txt')
>>> year, hares, lynxes, carrots = data.T # trick: columns to variables
```

Then plot it:

```
>>> from matplotlib import pyplot as plt
>>> plt.axes([0.2, 0.1, 0.5, 0.8])
>>> plt.plot(year, hares, year, lynxes, year, carrots)
>>> plt.legend(('Hare', 'Lynx', 'Carrot'), loc=(1.05, 0.5))
```

OUTPUT :-


80000
70000
60000
50000
40000
30000
20000
10000
0
1900
1905
1910
1915
1920
Hare
Lynx
Carrot