NumPy

- NumPy is a Python library.
- NumPy is used for working with arrays.
- NumPy is short for "Numerical Python".

Mr. Zohaib Ahmed
Software Engineering
Department Quest Nawabshah.
zohaibquest22@gmail.com

What is Numpy?

- NumPy is a Python library used for working with arrays.
- It also has functions for working in domain of linear algebra, fourier transform, and matrices.
- NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.
- NumPy stands for Numerical Python.

Why do we need NumPy

Let’s see for ourselves!

Why do we need NumPy

- In Python we have lists that serve the purpose of arrays, but they are slow to process.
- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
- Arrays are very frequently used in data science, where speed and resources are very important.

Which Language is NumPy written in?

- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

NumPy Overview

1. Arrays
2. Shaping and transposition
3. Mathematical Operations
4. Indexing and slicing
5. Broadcasting

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- **Vectors**
- **Matrices**
- Images
- Tensors
- ConvNets

$$\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}$$

$$\begin{bmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{bmatrix}$$

Arrays

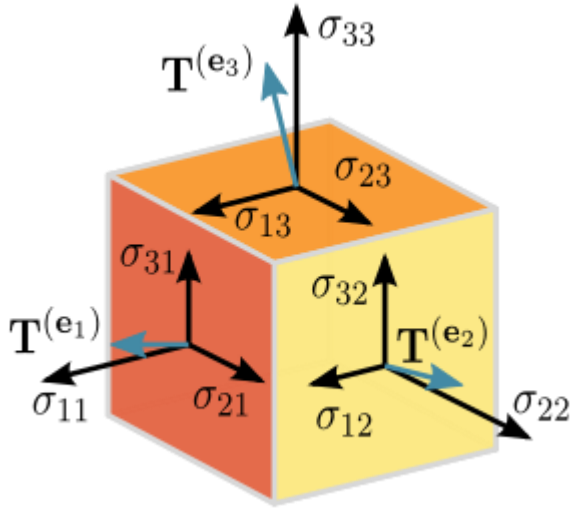
Structured lists of numbers.

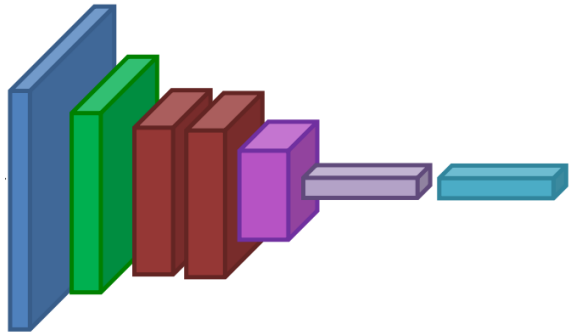
- Vectors
- Matrices
- **Images**
- Tensors
- ConvNets

Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- **Tensors**
- **ConvNets**





Arrays

Structured lists of numbers.

- Vectors
- Matrices
- Images
- Tensors
- ConvNets

Installation of NumPy

- If you have Python and PIP already installed on a system, then installation of NumPy is very easy.
- Install it using this command:

```
C:\Users\Your Name>pip install numpy
```

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

```
import numpy
```

Now NumPy is imported and ready to use.

Create a NumPy ndarray Object

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- We can create a NumPy ndarray object by using the array() function.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)

print(type(arr))
```

```
[1 2 3 4 5]
<class 'numpy.ndarray'>
```

Create a NumPy ndarray Object

- To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Example
Use a tuple to create a NumPy array:

```
import numpy as np

arr = np.array((1, 2, 3, 4, 5))

print(arr)
```

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

Dimensions in Arrays

- A dimension in arrays is one level of array depth (nested arrays).
- **nested array:** are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example

Create a 0-D array with value 42

```
import numpy as np

arr = np.array(42)

print(arr)
```

```
42
```

1-D Arrays

- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
- These are the most common and basic arrays.

```
# Example
# Create a 1-D array containing the values 1,2,3,4,5:

import numpy as np

arr = np.array([1, 2, 3, 4, 5])

print(arr)
```

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

2-D Arrays

- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.

```
# Example
# Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:
import numpy as np

arr = np.array([[1, 2, 3], [4, 5, 6]])

print(arr)
```

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

3-D arrays

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.
- These are often used to represent a 3rd order tensor.

```
# Example
# Create a 3-D array with two 2-D arrays, both containing two arrays
# with the values 1,2,3 and 4,5,6:

import numpy as np

arr = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(arr)
```

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

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

Check Number of Dimensions?

```
# Example
# Check how many dimensions the arrays have:

import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

Higher Dimensional Arrays

- An array can have any number of dimensions.
- When the array is created, you can define the number of dimensions by using the ndmin argument.

```
# Example
# Create an array with 5 dimensions and verify that it has 5 dimensions:

import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=5)

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

Arrays, creation

- **np.ones, np.zeros**
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```python
# Create a 1D array (vector) of ones
ones_vector = np.ones(5)
print("Ones Vector:")
print(ones_vector)

# Create a 2D array (matrix) of ones
ones_matrix = np.ones((3, 2))
print("\nOnes Matrix:")
print(ones_matrix)
```

```
Ones Vector:
[1. 1. 1. 1. 1.]

Ones Matrix:
[[1. 1.]
 [1. 1.]
 [1. 1.]]
```

Arrays, creation

- np.ones, np.zeros
- **np.arange**
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
array_1d = np.arange(10)
# output
# [0 1 2 3 4 5 6 7 8 9]

array_custom_range = np.arange(2, 10)
# output
# [2 3 4 5 6 7 8 9]

array_with_step = np.arange(0, 10, 2)
# output
# [0 2 4 6 8]
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
a = np.ones((2,3))
b = np.zeros((4,3))

np.concatenate([a,b])
```

```
array([[1., 1., 1.],
       [1., 1., 1.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.],
       [0., 0., 0.]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- **np.concatenate**
- np.astype
- np.zeros_like, np.ones_like
- np.random.random

```
a = np.ones((4,1))
b = np.zeros((4,2))

np.concatenate([a,b],axis=1)
```

```
array([[1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.],
       [1., 0., 0.]])
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- **np.astype**
- np.zeros_like, np.ones_like
- np.random.random

```
# Create an array of integers
original_array = np.array([1, 2, 3, 4, 5])

# Convert the array to float type
float_array = original_array.astype(float)

# Convert the array to string type
string_array = original_array.astype(str)
```

Original Array: [1 2 3 4 5]
Float Array: [1. 2. 3. 4. 5.]
String Array: ['1' '2' '3' '4' '5']

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- **np.zeros_like, np.ones_like**
- np.random.random

```
a = np.ones((2,2,3))
b = np.zeros_like(a)
```

```
b.shape
```

```
(2, 2, 3)
```

Arrays, creation

- np.ones, np.zeros
- np.arange
- np.concatenate
- np.astype
- np.zeros_like, np.ones_like
- **np.random.random**

```
np.random.random((10,3))
```

```
array([[0.93494113, 0.96507962, 0.72308189],
       [0.87376209, 0.24212469, 0.40897171],
       [0.0757972 , 0.73978703, 0.59173376],
       [0.93352748, 0.0011311 , 0.03267872],
       [0.67066181, 0.33771832, 0.57019079],
       [0.0721013 , 0.61725381, 0.05337661],
       [0.88474997, 0.08768563, 0.25577905],
       [0.07687186, 0.40943716, 0.83028458],
       [0.14694762, 0.29816206, 0.46619636],
       [0.26351157, 0.95130801, 0.33701517]])
```

Arrays, danger zone

- Must be dense, no holes.
- Must be one type
- Cannot combine arrays of different shape

```
>>> np.ones([7,8]) + np.ones([9,3])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: operands could not be broadcast together
 with shapes (7,8) (9,3)
```

Shaping

```
a = np.array([1,2,3,4,5,6])
a
```

```
array([1, 2, 3, 4, 5, 6])
```

```
a.reshape(3,2)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

```
a.reshape(2,-1)
```

```
array([[1, 2, 3],
       [4, 5, 6]])
```

```
a.ravel()
```

```
array([1, 2, 3, 4, 5, 6])
```

Transposition

```
a = np.arange(10).reshape(5,2)
a = a.T
```

Transposition

```
a = np.arange(10).reshape(5,2)
a = a.T
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

Transposition

```
a = np.arange(10).reshape(5,2)
```

```
a = a.T
```

```
[[0 1]
 [2 3]
 [4 5]
 [6 7]
 [8 9]]
```

```
[[0 2 4 6 8]
 [1 3 5 7 9]]
```

Saving and loading arrays

```
np.savez('data.npz', a=a)
data = np.load('data.npz')
a = data['a']
```

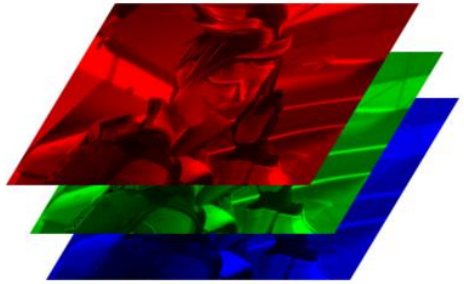
1. NPZ files can hold multiple arrays
2. np.savez_compressed similar.

Image arrays

Images are 3D arrays: width, height, and channels

Common image formats:

- height x width x RGB (band-interleaved)
- height x width (band-sequential)



Gotchas:

- Channels may also be BGR (OpenCV does this)
- May be [width x height], not [height x width]

Saving and Loading Images

SciPy: `skimage.io.imread`,`skimage.io.imsave`

height x width x RGB

PIL / Pillow: `PIL.Image.open`, `Image.save`

width x height x RGB

OpenCV: `cv2.imread`, `cv2.imwrite`

height x width x BGR

Recap

We just saw how to create arrays, reshape them, and permute axes

Questions so far?

Recap

We just saw how to create arrays, reshape them, and permute axes

Questions so far?

Now: let’s do some math

Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- In place operations modify the array

Mathematical operators

- **Arithmetic operations are element-wise**
- Logical operator return a bool array
- In place operations modify the array

```
>>> a
array([1, 2, 3])
>>> b
array([ 4,  4, 10])
>>> a * b
array([ 4,  8, 30])
```

Mathematical operators

- Arithmetic operations are element-wise
- **Logical operator return a bool array**
- In place operations modify the array

```
>>> a
array([[ 0.93445601,  0.42984044,  0.12228461],
       [ 0.06239738,  0.76019703,  0.11123116],
       [ 0.14617578,  0.90159137,  0.89746818]])
>>> a > 0.5
array([[ True, False, False],
       [False,  True, False],
       [False,  True,  True]], dtype=bool)
```

Mathematical operators

- Arithmetic operations are element-wise
- Logical operator return a bool array
- **In place operations modify the array**

```
>>> a
array([[ 4, 15],
       [20, 75]])
>>> b
array([[ 2,  5],
       [ 5, 15]])
>>> a /= b
>>> a
array([[2, 3],
       [4, 5]])
```

Math, universal functions

Also called ufuncs

Element-wise

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

Math, universal functions

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
import numpy as np

x = np.array([1, 2, 3])
result = np.exp(x)

print(result)
```

```
[ 2.71828183  7.3890561  20.08553692]
```

Math, universal functions

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
import numpy as np

x = np.array([4, 9, 16])
result = np.sqrt(x)

print(result)
```

```
[2. 3. 4.]
```

Math, universal functions

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
import numpy as np

x = np.array([0, np.pi/2, np.pi])
result = np.sin(x)

print(result)
```

```
[0.0000000e+00 1.0000000e+00 1.2246468e-16]
```

Math, universal functions

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
import numpy as np

x = np.array([0, np.pi/2, np.pi])
result = np.cos(x)

print(result)
```

```
[ 1.000000e+00  6.123234e-17 -1.000000e+00]
```

Math, universal functions

Examples:

- np.exp
- np.sqrt
- np.sin
- np.cos
- np.isnan

```
import numpy as np

x = np.array([1.0, np.nan, 3.0])
result = np.isnan(x)

print(result)
```

```
[False  True False]
```

Indexing

```
x[0,0]    # top-left element
x[0,-1]   # first row, last column
x[0,:]    # first row (many entries)
x[:,0]    # first column (many entries)
```

Assuming x is a 2D array

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

Python Slicing

Syntax: start:stop:step

```
a = list(range(10))
a[:3] # indices 0, 1, 2
a[-3:] # indices 7, 8, 9
a[3:8:2] # indices 3, 5, 7
a[4:1:-1] # indices 4, 3, 2 (this one is tricky)
```

Axes

```
a.sum() # sum all entries
a.sum(axis=0) # sum over rows
a.sum(axis=1) # sum over columns
a.sum(axis=1, keepdims=True)
```

```
# Assuming a is a 2D array
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Try yourself….

Broadcasting

```
Array a:
[[1 2 3]
 [4 5 6]]

Array b:
[10 20 30]

Result of a + b:
[[11 22 33]
 [14 25 36]]
```

```
import numpy as np

# Example arrays
a = np.array([[1, 2, 3],
              [4, 5, 6]])

b = np.array([10, 20, 30])

# Broadcasting in action
result = a + b

print("Array a:")
print(a)

print("\nArray b:")
print(b)

print("\nResult of a + b:")
print(result)
```

Here's a quick breakdown of the shapes:

- Array a has shape (2, 3) because it is a 2x3 matrix.
- Array b has shape (3,) because it is a 1D array with 3 elements.

Since the smaller array b is broadcasted to the shape of the larger array a during the addition operation, the resulting array will have the shape of a, which is (2, 3).

NumPy

Linked in kaggle YouTube GitHub

Mr. Zohaib Ahmed
Software Engineering
Department Quest Nawabshah.
zohaibquest22@gmail.com

ANACONDA jupyter