Introduction to Numpy

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object. In this course, we will be using NumPy for linear algebra.

If you are interested in learning more about NumPy, you can find the user guide and reference at https://docs.scipy.org/doc/numpy/index.html (https://docs.scipy.org/doc/numpy/index.html)

Let's first import the NumPy package

In [1]:
```
import numpy as np # we commonly use the np abbreviation when referring to numpy
```

Creating Numpy Arrays

New arrays can be made in several ways. We can take an existing list and convert it to a numpy array:

In [2]:
```
a = np.array([1,2,3])
a
```

Out[2]:
```
array([1, 2, 3])
```

There are also functions for creating arrays with ones and zeros

In [3]:
```
np.zeros((2,2))
```

Out[3]:
```
array([[ 0.,  0.],
       [ 0.,  0.]])
```

In [4]:
```
np.ones((3,2))
```

Out[4]:
```
array([[ 1.,  1.],
       [ 1.,  1.],
       [ 1.,  1.]])
```

Accessing Numpy Arrays

You can use the common square bracket syntax for accessing elements of a numpy array

In [5]:
```
A = np.arange(9).reshape(3,3)
print(A)
```

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

In [32]:
```
print(A[0]) # Access the first row of A
print(A[0, 1]) # Access the second item of the first row  also A[0][1] works
print(A[:, 1]) # Access the second column
print(A[:,1][1])    #Access the second item of the second column
```

```
[0 1 2]
1
[1 4 7]
4
```

Operations on Numpy Arrays

You can use the operations '*', '**', '\', '+' and '-' on numpy arrays and they operate elementwise.

In [33]:
```
a = np.array([[1,2],
              [2,3]])
b = np.array([[4,5],
              [6,7]])
```

In [40]:
```
#matrix addition
print(a + b)
```

```
[[ 5  7]
 [ 8 10]]
```

In [41]:
```
#matrix subtraction
print(a - b)
```

```
[[-3 -3]
 [-4 -4]]
```

In [43]:
```
#matrix multiplication
print(a@b)
```

```
[[16 19]
 [26 31]]
```

In [44]:
```
#element-wise multiplication
print(a * b)
```

```
[[ 4 10]
 [12 21]]
```

In [45]:
```
print(a / b)
```

```
[[ 0.25       0.4       ]
 [ 0.33333333  0.42857143]]
```

In [46]:
```
print(a**2)
```

```
[[1 4]
 [4 9]]
```

There are also some commonly used function For example, you can sum up all elements of an array

In [47]:
```
print(a)
print(np.sum(a))
```

```
[[1 2]
 [2 3]]
8
```

Or sum along the first dimension

In [54]:
```
#row-wise addition
np.sum(a, axis=0)

# np.sum([[1, 3], [5, 4]], axis=1) [4, 9] column-wise addition
```

Out[54]:
```
array([3, 5])
```

There are many other functions in numpy, and some of them **will be useful** for your programming assignments. As an exercise, check out the documentation for these routines at https://docs.scipy.org/doc/numpy/reference/routines.html (https://docs.scipy.org/doc/numpy/reference/routines.html) and see if you can find the documentation for `np.sum` and `np.reshape`.

Linear Algebra

In this course, we use the numpy arrays for linear algebra. We usually use 1D arrays to represent vectors and 2D arrays to represent matrices

In [55]:
```
A = np.array([[2,4],
              [6,8]])
```

You can take transposes of matrices with `A.T`

In [56]:
```
print('A\n', A)
print('A.T\n', A.T)
```

```
A
 [[2 4]
 [6 8]]
A.T
 [[2 6]
 [4 8]]
```

Note that taking the transpose of a 1D array has **NO** effect.

In [57]:
```
a = np.ones(3)
print(a)
print(a.shape)
print(a.T)
print(a.T.shape)
```

```
[ 1.  1.  1.]
(3,)
[ 1.  1.  1.]
(3,)
```

But it does work if you have a 2D array of shape (3,1)

In [58]:
```
a = np.ones((3,1))
print(a)
print(a.shape)
print(a.T)
print(a.T.shape)
```

```
[[ 1.]
 [ 1.]
 [ 1.]]
(3, 1)
[[ 1.  1.  1.]]
(1, 3)
```

Dot product

We can compute the dot product between two vectors with np.dot

In [60]:
```
x = np.array([1,2,3])
y = np.array([4,5,6])
np.dot(x, y)
```

Out[60]: 32

We can compute the matrix-matrix product, matrix-vector product too. In Python 3, this is conveniently expressed with the @ syntax

In [61]:
```
A = np.eye(3) # You can create an identity matrix with np.eye
B = np.random.randn(3,3)
x = np.array([1,2,3])
```

In [68]:
```
B
```

Out[68]:
```
array([[-0.48250102,  1.15098332, -1.02581822],
       [ 1.49749632, -0.72628172, -0.74093888],
       [-1.14076646,  1.30340195,  1.15492831]])
```

In [64]:
```
#identity matrix of shape nxn
A
```

Out[64]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

In [70]:
```
# Matrix-Matrix product
A @ B
#matrix @ identity matrix = same matrix
```

Out[70]:
```
array([[-0.48250102,  1.15098332, -1.02581822],
       [ 1.49749632, -0.72628172, -0.74093888],
       [-1.14076646,  1.30340195,  1.15492831]])
```

In [71]:
```
# Matrix-vector product
A @ x
```

Out[71]:
```
array([ 1.,  2.,  3.])
```

Sometimes, we might want to compute certain properties of the matrices. For example, we might be interested in a matrix's determinant, eigenvalues/eigenvectors. Numpy ships with the `numpy.linalg` package to do these things on 2D arrays (matrices).

In [72]:
```
from numpy import linalg
```

In [73]:
```
# This computes the determinant
linalg.det(A)
```

Out[73]:
```
1.0
```

In [74]:
```
# This computes the eigenvalues and eigenvectors
eigenvalues, eigenvectors = linalg.eig(A)
print("The eigenvalues are\n", eigenvalues)
print("The eigenvectors are\n", eigenvectors)
```

```
The eigenvalues are
 [ 1.  1.  1.]
The eigenvectors are
 [[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

Miscellaneous

Time your code

One tip that is really useful is to use the magic commannd `%time` to time the execution time of your function.

In [75]:
```
%time np.abs(A)
```

```
CPU times: user 10 µs, sys: 5 µs, total: 15 µs
Wall time: 17.6 µs
```

Out[75]:
```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```