Open in Colab

Numpy (Array-Like Data Structures)

Resources:

- Numpy Documentation

Numpy

Short for "numeric python", this library is **built around storing data in arrays and performing numeric operations on items in the arrays**. Arrays can be **two-dimensional** like a list [0, 1, 2] *or* they can also be **multi-dimensional** like lists of lists or lists of lists of lists, etc.

The standard nickname convention for numpy is to import it as **np**.

In [3]:
```
# can shorten module names
import numpy as np
```

Numpy has support for converting several common data types to arrays. **Below, we**
are converting the python list [1,2,3,4,5] to a numpy array with the same values. Notice how it prints as array([1,2,3,4,5]).

In [4]:
```
numpy_array = np.array([1,2,3,4,5])
numpy_array
```

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

Numpy has better support for doing numeric operations than lists do. For example, **we can easily add a value to every item in the array** at once.

In [5]:
```
numpy_array + 1
```

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

Similar to the **range()** function, numpy has a function called **arange** which creates an array of [0,1,2,...,N].

In [6]:
```
size = 1000000
```

```
# declaring arrays
array1 = np.arange(size)
array2 = np.arange(size)
```

If we multiply arrays by one another, **items of matching indices will multiply**. Below we are multiplying all items from indices 1 to 10-1=9. Notice how these **indices start at 0 and work similarly to list indices**.

In [7]:
```
array1[1:10] * array2[1:10]
```

Out[7]:
```
array([ 1,  4,  9, 16, 25, 36, 49, 64, 81])
```

We can also use operations like **dot()** which **takes the dot product**.
Reminder: [2, 7, 9] dottted with [3, 5, 1] equals 2 * 3 + 7 * 5 + 9 * 1

In [8]:
```
# dot product
np.dot(array1[1:10], array2[1:10])
```

Out[8]:
```
np.int64(285)
```

A **matrix** is a name for **an array that has multiple dimensions**. Below is an example of a 3x3 matrix.

In [11]:
```
# matrices
matrix = np.array([[1,2,3],[4,5,6],[7,8,0]])
matrix
```

Out[11]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 0]])
```

We can also **select items from an array or matrix using indices** like we did with lists.

matrix[0] gets the first row
matrix[0,0] get the first column from the first row
matrix[:,0] gets items in all rows but only for the first column
The symbol : on its own means get all items.

In [12]:
```
# numpy arrays
print(matrix[0])
print(matrix[0,0])
print(matrix[:,0])
```

```
[1 2 3]
1
[1 4 7]
```

Numpy does not really care about row/column vectors. In general **numpy can automatically pick axes that make the most sense for your operation**. Be careful with this as it means numpy will sometimes successfully do something even if it's not what you intended. Here's an example of 3x3 dot products.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 8 \\ 17 \\ 8 \end{pmatrix} \tag{1}$$

$$\begin{pmatrix} 0 & 1 & 2 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{pmatrix} = \begin{pmatrix} 18 & 21 & 6 \end{pmatrix} \tag{2}$$

In [13]:
```
# matrix vector multiplication
print(np.dot(matrix, array1[0:3])) #
print(np.dot(array1[0:3], matrix))
```

```
[ 8 17  8]
[18 21  6]
```

In [14]:
```
# matrix matrix multiplication
np.dot(matrix, matrix)
```

Out[14]:
```
array([[30, 36, 15],
       [66, 81, 42],
       [39, 54, 69]])
```

Numpy as has a library called "random" that's great for generating different types of random data. What the below code says is, **using the random integer (randint) function** from the random library in numpy, I want to **create random integers with values less than 10 until I've filled up a 3x4x5 array**.

In [15]:
```
array_3D = np.random.randint(10, size=(3,4,5))
array_3D  # 3D matrix
```

Out[15]:
```
array([[[9, 3, 6, 9, 8],
        [8, 0, 7, 2, 9],
        [8, 1, 1, 9, 8],
        [1, 0, 5, 0, 1]],

       [[5, 7, 6, 7, 7],
        [2, 8, 0, 9, 1],
        [2, 5, 6, 9, 0],
        [2, 7, 2, 8, 1]],

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

ndim tells us how many dimensions/axes the array has

In [16]:
```
array_3D.ndim
```

Out[16]:
```
3
```

shape tells us the exact size of those dimensions/axes in order

In [17]:
```
array_3D.shape # somethin python lists can't do
```

Out[17]:
```
(3, 4, 5)
```

$$M_{ijk} \rightarrow M_{ikj} \tag{3}$$

Transpose is a very important function to know for machine learning. Often data doesn't come in the exact format or order that you want it in. Being able to **change the order of the axes** is important. Here we're taking axes [0,1,2] and reordering them like [0,2,1]. In other words, we're swapping our last two axes.

In [18]:
```
np.transpose(array_3D, axes=[0,2,1])
```

Out[18]:
```
array([[[9, 8, 8, 1],
        [3, 0, 1, 0],
        [6, 7, 1, 5],
        [9, 2, 9, 0],
        [8, 9, 8, 1]],

       [[5, 2, 2, 2],
        [7, 8, 5, 7],
        [6, 0, 6, 2],
        [7, 9, 9, 8],
        [7, 1, 0, 1]],

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

We can also use **.reshape()** to swap axes or **change the shape of our data**. This is often **less safe than transpose** because it can accept shapes that don't make much sense given your data. In other ways it **can be more useful like taking 9 items and reshaping it as 3x3 which you can't do with transpose**. Be careful when using this.

In [19]:
```
array1[0:5]
```

Out[19]:
```
array([0, 1, 2, 3, 4])
```

In [20]:
```
array1[0:5].reshape(5,1)
```

Out[20]:
```
array([[0],
       [1],
       [2],
       [3],
       [4]])
```

In [21]:
```
matrix
```

Out[21]:
```
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 0]])
```

Sometimes we may want to flatten our matrix. We can do this with reshape or the flatten function.

In [22]:
```
matrix.reshape(9)
```

Out[22]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 0])
```

In [23]:
```
matrix.flatten()
```

Out[23]:
```
array([1, 2, 3, 4, 5, 6, 7, 8, 0])
```

Just like how **we could append, delete and overwrite list values**, we can do similar operations with numpy arrays.

In [24]:
```
array3 = np.array([3,5,7])
```

In [25]:
```
np.append(array3, 1)  # does not change list3, but returns new list
```

Out[25]:
```
array([3, 5, 7, 1])
```

In [26]:
```
array3 = np.append(array3, [1,2,3])
array3
```

Out[26]:
```
array([3, 5, 7, 1, 2, 3])
```

Below we show the delete function. Here we're using it to take array 3, and return a copy of it with the value at index 1 removed.

In [27]:
```
np.delete(array3, 1)  # delete second element. Also does not change ar
```

Out[27]:
```
array([3, 7, 1, 2, 3])
```

Since **numpy functions like append and delete make copies of numpy arrays**, you
need to save them to a new array or overwrite your old array to save them.

In [28]:
```
array3
```

Out[28]:
```
array([3, 5, 7, 1, 2, 3])
```

Before getting to an activity, let's review some concepts

We can create a function using:
def func_name(parameter1, parameter2):

We can then use that function in the following way:
func_name(a, b)

In [29]:
```
def function(parameter1, parameter2):
  print("do something")
function(1, "A")
```

```
do something
```

We have reviewed for loops and how we can use them to iterate but there is also a type of loop called a while loop. **While loops execute code inside the loop *while* a statement is True**. The following code says, while num is not equal to 1, subtract 1 from it then print it.

In [30]:
```
num = 10
while num != 1:
  num -= 1
  print(num)
```

```
9
8
7
6
5
4
3
2
1
```

The **% or modulo function gets the remainder of division**. This is often **used to execute a function only every so many steps**. For example, in the following code, we're printing only every 1,000 steps. The exact statement is, **if the step number divided by 1000 has a remainder of 0, then print the step number**.

In [31]:
```
for i in range(5000):
  if i % 1000 == 0:
    print(i)
```

```
0
1000
2000
3000
4000
```

We can also use modulo to check other numeric properties such as even or odd by
by checking the remainder of dividing by 2.

In [32]:
```
for i in range(6):
  if i % 2 == 1:
    print(i)
```

```
1
3
5
```

Arrays can be constructed from lists or from single items.

In [33]:
```
arr1 = np.array(1)
arr2 = np.array([1,2])
print("arr1:", arr1)
print("arr2:", arr2)
```

```
arr1: 1
arr2: [1 2]
```

If we append an item to an array, it will create a new array which is a copy of the original array with the new item appended. We can add an item to a list by appending in place in the following way:

In [34]:
```
print(np.append(arr1, 2))
print("arr1 after running append without assignment:", arr1)
arr1 = np.append(arr1, 2)
print("arr1 after running append with assignment:   ", arr1)
```

```
[1 2]
arr1 after running append without assignment: 1
arr1 after running append with assignment:    [1 2]
```

Activity

The Collatz conjecture is a simple math statement. It says, start from any number. If that number is odd, multiply it by 3 then add 1. If it's even, divide it by 2. Eventually that number will go to 1.

Example:
3 * 3 + 1 = 10
10 / 2 = 5
5 * 3 + 1 = 16
16 / 2 = 8 => 4 => 2 => 1

Create a function called collatz(start) which will take a number called start and execute the Collatz conjecture until it reaches 1. For each step, append it to the end of an array.

Below, we'll review some concepts you might needfor this activity.

Required code:

```
def collatz(number):
    array = np.array(number)
    while {write an ending condition here}:
        if number % 2 == 0:
            {write your collatz operation for even numbers}
        else:
            {write your collatz operation for odd numbers}
        {write a statement appending your current value to
array}
    print(array)
collatz(7)
```

In [85]:
```
#Your code goes here:
########################
def collatz(number):
    array = np.array([number])
    while array[-1]>1:
        if number % 2 ==0:
            number = number/2
        else:
            number = number*3+1
        array = np.append(array,number)
    print(array)

########################
```

In [86]:
```
collatz(7)
```

```
[ 7. 22. 11. 34. 17. 52. 26. 13. 40. 20. 10.  5. 16.  8.  4.  2.  1.]
```

In [ ]:
```
#EXAMPLE OUTPUT
```

```
[ 7. 22. 11. 34. 17. 52. 26. 13. 40. 20. 10.  5. 16.  8.  4.  2.  1.]
```

In [ ]: