# Chapter 9

# Numpy: Review

Numpy short for numerical Python , is a Python package that is used for multidimensional array manipulation and operations. It is a library for scientific computing in Python . Numpy operations are very efficient in terms of time and memory and speed up the processing by quite a large margin. Below we will present some basic numpy codes for a quick introduction of this useful Python library [1].

```
1 import numpy as np
2 print(np.__version__)
```

The convention is to import numpy module with np alias. We can use the functions, definitions and statements etc in numpy module using the `.` operator. For example in the above code we access the `__version__` to print the version of numpy.

```
print(np.array(3).shape)
print(np.array([3]).shape)
print(np.array([[3]]).shape)
print(np.array([[[3]]]).shape)
```

Numpy arrays are the primary data processing containers in numpy. Numpy arrays are defined by using np.array command. The [] brackets are used to group data into vectors and matrices. The above code has same data in all arrays but the shape of the arrays is different owing to the [] we have used in each line of code.

```
import time as time
x_list=[i for i in range(100000)]
x=np.array(x_list)

st=time.process_time()
x_list=[i+10 for i in x_list]
ed=time.process_time()
print(round(ed-st,5))


st=time.process_time()
x+=10
ed=time.process_time()
print(round(ed-st,5))
```

[1] Parts of this chapter is motivated from https://cs231n.github.io/python-numpy-tutorial/

One of the main advantage of numpy arrays over python containers like lists etc is it's speed. Operations can be performed on numpy arrays much faster than on Python lists etc. The above code times the increment by constant operation for numpy arrays and Python lists and displays the time. Notice that for numpy arrays the operation is much faster.

```
x=np.array([1,2,3,4])
print(type(x))
print(x[0],x[1],x[2],x[3])
x[1]=x[2]
print(x)
print(x.shape)

y=np.array([[1,2,3],[4,5,6],[7,8,9]])
print(y.shape)
print(y.size)
print(y[1,2],y[0,1], y[2,2])

#indexing with negative indices
print(y[1,-1])
```

Numpy arrays can be initialized with multidimensional (or one dimensional) Python lists. All elements of the numpy array are of the same data type and the data type can be accessed with the type command of Python . Numpy array are indexed in the form x[obj] where x is the numpy array and obj it the type of indexing which can be field access, basic slicing and advanced slicing.

```
x=np.random.random((4,3))
print(x)

print('printing first column of x', x[:,0:1])
print('printing first row of x', x[0:1,:])
print('printing ? row ? cols', x[:1,2:])
print('printing ? row ? cols', x[1:3,1:3])
print('printing ? row ? cols', x[1:2,1:2])

y=x[1:3,1:3]
y[0,0]=0
#slice is a view into original array modifying it will
#modify original array as well
print(x)
```

Using index we can access rows or columns of the numpy array and even multiple rows and columns of the numpy array. The above code extracts rows and columns and specific combinations of rows and columns of numpy arrays. Here, we would like to convey some important distinctions between integer slicing and basic slicing. The syntax for basic slicing for numpy arrays is start:stop[:step] for each dimension. The slice obtained will have the elements in the corresponding dimension starting from "start" with step size of "step" and ending one place before "end". "start" and "step" argument are optional and by default their value is None. Basic slicing in Python gets a slice of the original array without copying it to a new memory location. Any changes made to the slice will result in changes in the array as well.

On the other hand integer indexing gets us a copy of the indices we have specified, and changes to them do not effect the original array.

```
#difference of integer indexing and slicing

x=np.random.random((5,5))
print(x)
y1=x[:,1]
y2=x[:,1:2]
print(y1,y1.shape)
print(y2,y2.shape)


z1=x[:,1]
z2=x[:,1:2]
print(z1,z1.shape)
print(z2,z2.shape)


print(x[[1,2,3],:].shape)
```

The above codes shows the difference of integer indexing and basic slicing for numpy arrays. Notice that integer indexing and basic slicing are accessing the same elements in the array but the output of integer indexing is a one dimensional array and the output of the basic slicing has the same number of dimensions as the original array albeit the size of each dimension is fixed by the slicing operation.

**Note:** Notice that a combination of integer slicing and basic slicing in numpy will give you a slice from the main array and changing it will cause changes in the main array.



```
#integer slicing tricks: one use of the integer index is to
#modify an elements in rows or columns on an np array

x=np.random.random([5,3])
#let's divide an element for each row by 2, first
row_ind=[1,0,2,0,1]
r,c= x.shape
print(x[np.arange(r),row_ind])
print(x)
x[np.arange(r),row_ind]/=2
print(x)

#let's multiply an element from each column by 5

col_ind=[3,4,1]
x[col_ind,np.arange(c)]*=5
print(x)
```

The above code uses integer slicing to modify a specific element in each row or column of the the array. Of course a similar scheme can be used to modify any random subset of elements of an array as long as we know the the indices of all the elements that needs to be changed.

```
x=np.random.random((6,6))
y=x[[1,2,1,3,0],[2,3,4,5,1]]

print(y)
print (x)
y[0]=1
print(x)
print(y)
```

As suggested in the note above integer indexing generates a new array as opposed to getting a slice from the original array so changing the new array does not change elements in the original array.

```
x=np.random.random([3,3])
sel_ind=x>0.5
print(sel_ind)
print(x[sel_ind])
```

In many applications we need to extract certain elements from a numpy array that satisfies a certain condition. For example we might be interested to find the days of the month in December where temperature was below zero degrees. This can be done in numpy with Boolean indexing. Boolean indexing essentially translates to a True of False value for an index, if the value is True the element at that index is selected else it is not.

**Note:** Similar to integer indexing Boolean indexing create a copy of the array elements and changing the copy does not change the original array.



```
x = np.zeros((3,2))
print(x)

y = np.ones((2,2))
print(y)

iden = np.eye(2)
print(iden)

r = np.random.random((2,2))
print(r)

c = np.full((2,2), 3)
print(c)
```

Many a times in programming exercises we need to generate arrays with particular type of values. For example zeros or ones, identity matrix or matrix with random values sampled from a particular distribution. Numpy has functions for generating these types of matrices. We can generate metrics with fixed values, identity matrices and random matrices with values distributed from uniform and normal distributions and random integers uniformly distributed from a range.

### Numpy Datatypes

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

Numpy allows us to use arrays of multiple datatypes. The ability to work with multiple datatypes is very useful. Imagine two applications where in one application you have to assign a unique ID to each student in a class and in the other application you are recording the temperatures observed in a city each day. In the first application you will probably not need Real numbers and positive integers would be sufficient for your need, where as in the second application you will need to work with real numbers. Using real numbers in the first application will add useless complexity to your problem and using integers in the second example will result in data loss. Therefore, it is nice to have the ability to use multiple datatype while programming the solution for a particular problem.

```
x=np.random.randint(0,9,(1,4))
y=np.random.randint(2,5,(1,4))
print(x,y)
print(x+y)
print(np.add(x,y))

print(x-y)
print(np.subtract(x,y))

print(x*y)
print(np.multiply(x,y))

print(x/y)
print(np.divide(x,y))

print("I have intentionally started the range of y from value
greater than zero, Can you think or a reason why?")

print(np.sqrt(x),np.sqrt(y))
```

The above code provides some of the operations we can perform on numpy array. Notice that all of the above operations are element-wise operations in numpy and performed element wise. $*$ is not matrix multiplication rather element wise multiplication between elements of the two arrays.

**Good Practice:** Always make sure that there are no divisions by zero in your code!

```
x=np.random.randint(0,9,(4,))
y=np.random.randint(0,9,(4,))


print(x.shape, y.shape)

print(x.dot(y.T))
```

```
print(np.dot(x,y.T))

print(np.matmul(x,y))

x_f=np.random.randint(0,9,(1,4))
y_f=np.random.randint(0,9,(4,1))


print(x.shape, y.shape)
print(x_f.dot(y_f))
print(np.dot(x_f,y_f))
print(np.matmul(x_f,y_f))


w=np.random.randint(0,9,(3,4))
z=np.random.randint(0,9,(4,3))

print(w.dot(z))
print(np.dot(w,z))
print(np.matmul(w,z))
```

For matrix multiplication (2D arrays) in numpy we can use either *np.dot* or *np.matmul* (np.matmul is preferred). Notice that for using matrix multiplications the two arrays should be compatible i.e. the number of columns of the first array should be equal to the number or rows of the second array. For 1D arrays np.dot returns the dot product. If both arrays in np.dot are multi-dimensional np.dot returns the product sum over the last dimension of the first array and second to last dimension of the second array.

**Note:** For complex conjugate dot product use np.vdot.

```
x=np.random.randint(0,9,(3,4,2))

print("x is :",x)
print("sum of x is:", np.sum(x))
print(np.sum(x,axis=0))
print(np.sum(x,axis=1))
print(np.sum(x,axis=2))


print("max of x is:",np.max(x))
print(np.max(x,axis=0))
print(np.max(x,axis=1))
print(np.max(x,axis=2))

print("min of x is:",np.min(x))
print(np.min(x,axis=0))
print(np.min(x,axis=1))
print(np.min(x,axis=2))
```

The above command demonstrates how to get the sum of a numpy array and the sum along a particular dimension of an array. Similarly we can also get the maximum or minimum values of a numpy array and maximum/minimum values along particular dimensions.

```
x=np.random.randint(0,9,(2,2))
y=np.random.randint(0,9,(2,2))
print(x)
print(y)
np.copyto(x,y)
print(x)
print(y)
```

**Note:** If we want to make a copy of a numpy array which can be modified independently of the original array, we can use np.copyto command.



```
x=np.random.randint(0,9,(4,3))
y=np.reshape(x,(2,6))
print(x)
print(y)

print(np.ravel(x,order='C'))    #try order 'C','F', 'A', 'K'

v=x.flat
print(v[6])

u=x.flatten()
print(u)
```

If we want to reshape an array we can use np.reshape. The size of the original array and the reshaped array should be same (by size we mean the number of elements), you can check the number of elements of a numpy array by using the command np.ndarray.size, you will see this type of notation quite a lot in numpy documentation what it means is the size is and attribution of ndarray which is a numpy object.

```
v=np.random.randint(0,9,(1,4))
print(v,v.T)
x=np.random.randint(0.9,(2,3))
print(x)
print(x.T)

z=np.zeros((2,3,4))
print(z.shape)
print(np.moveaxis(z,0,1).shape)
print(np.moveaxis(z,0,2).shape)


w=np.ones((5,3,4))
print(w.shape)
print(np.swapaxes(w,0,2).shape)

print(np.transpose(w, axes=(2,1,0)).shape)
print(np.transpose(w, axes=(0,1,2)).shape)
```

Transposition or moving dimensions is an important operation for array manipulation and comes in quite handy when you are doing matrix calculus. Numpy has transpose and swapaxes functions that can be used for these type of operations.

Complete list of array manipulation can be found here https://docs.scipy.org/doc/numpy/reference/routines.array-manipulation.html

### Numpy Broadcasting

Broadcasting is a tool in numpy which allows for operation on numpy arrays without the need for looping over all the elements of the numpy array. Starting from the trailing dimensions of the numpy arrays the rules for numpy broadcasting are as follows:

- The size of the $n^{th}$ trailing dimension for both arrays is equal
- The size of the $n^{th}$ trailing dimension for one of the arrays is equal to one
- if at least one the above two condition is satisfied for all the trailing dimensions where shape is non-zero for both arrays. The array with larger shape dimensions can have any value where the smaller array has zero shape.

Example: if we have two arrays of shape $l \times r \times d \times c$ and $c$. The arithmetic operations between them is compatible and the output shape will be $l \times r \times d \times c$.

Example: if we have two arrays of shape $l \times r \times 1 \times c$ and $l \times 1 \times d \times c$. The arithmetic operations between them is compatible and the output shape will be $l \times r \times d \times c$.

The dimension that is 1 is copied the number of time corresponding to the size on the same dimension of the other array.

```
import numpy as np

x=np.ones((1,2,3))
y=0.5*np.ones((3))

print(x*y)
print((x*y).shape)


x=np.ones((1,2,3))
y=0.5*np.ones((5,1,3))

print(x*y)
print((x*y).shape)
```

## 9.1 Plotting in Python

For plotting in Python we use MatPlotLib library. Below are a few sample codes explaining the use of matplotlib .

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline

x = np.linspace(-1, 1, 100)
y = x*x + 1
```

```
plt.plot(x, y)
plt.show()
```

```
import matplotlib.pyplot as plt
import numpy as np

%matplotlib inline


x = np.linspace(-1, 1, 100)

for i in range(6):
  y = i*x + 1
  plt.subplot(2,3,i+1)
  plt.plot(x, y)
plt.show()
```

### Exercises

**Exercise 9.1:** Time different operations for numpy arrays and `Python` lists and display the output.

**Exercise 9.2:** If you are given a random variable $x \sim U[0,1]$ can you generate a random variable $y \sim U[a,b]$ where $b > a$?

**Exercise 9.3:** Test np.dot(x,y) and np.matmul(x,y) on the following arrays.

- x and y are 1D
- x is 2D y is 1D
- x is 1D and 2D y is scalar
- x is 3D and y is 1D
- x is 3D and y is 3D

**Exercise 9.4:** Write a code which takes in the row and columns wise maximum and minimum values of two 2D arrays of same size and returns the upper and lower bounds of the maximum and minimum values of the sum of the two arrays.

**Exercise 9.5:** Try the following:

- np.transpose for 1D and 2D array
- np.transpose for 3D and 4D array
- np.swapaxes for 2D and 3D array

**Exercise 9.6:** Write a program that returns the Frobenius norm of a square matrix.

**Exercise 9.7:** Write a program to implement Newotn method to find the solution of $p(x,n)=0$, where $n$ is the order of the polynomial $p$. The input of the program is an array of the coefficent of the polynomials. Output is the solution $x_f$ of the equation. Example: input $[2,1,6]$ mean's we have to solve $2x^2+x+6=0$. Newton method is an iterative technique for solving non-linear equations with $x_{n+1}=x_n-\frac{f(x_n)}{f'(x_n)}$

**Exercise 9.8:** Write a program that takes in image (I) of shape $l\times w\times c$ and mask (m) of $l\times w$, where the mask has elements in $R=\{1,2,...,N\}$. We need to calculate $E=\sum_c\sum_{r\in R}\sum_{i,j\in l,w}(I[i,j,c])-a[r,c])^2*(m[i,j]==r)$. Where $a[r,c]=\sum_{i,j\in l,w}I[i,j,c])*(m[i,j]==r)/\sum_{i,j\in l,w}(m[i,j]==r)$. Do the above using for loops and without using for loops using numpy broadcasting.

Use the following code to generate the image and the mask.

```
im=np.zeros((32,32))
im[:16,:]=1

mask=np.zeros((32,32))
mask[:,:16]=1

plt.imshow(mask)
```