Chapter 13 - NumPy

13 NumPy

NumPy is a general purpose array processing package that provides tools for working with multidimensional array objects. This package is a fundamental package for scientific computing in Python. The NumPy library is called `numpy` and it's normally imported as `import numpy as np`. The `as np` is optional and is used to give a shorter name to the package.

13.1 Arrays

Arrays are NumPy's main objects. They are homogeneous (of the same type) and multidimensional. A single dimensional array is similar to a Python `List` in the way that it also uses **indexing** to access elements but also different because it can only store homogeneous datatypes. A 2 dimensional array is like a table with rows and columns and a 3 dimensional array is like a Rubik's Cube.

Terms to note:

- The elements are all of the same type and indexed by a tuple of positive integers
- The dimensions are called **axes** and the number of axes is called the **rank**
- The array class is called **ndarray** (alias array)

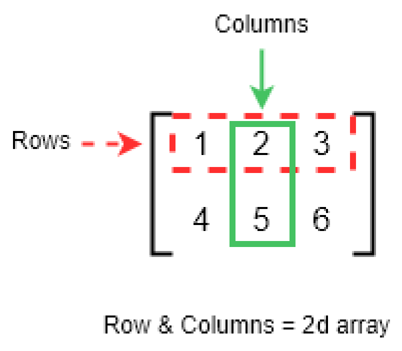

Figure 1: A 2 dimensional array.

The above figure 1 is a 2d array of rank 2 (because it has 2 dimensions). The first dimension number is the rows and the second dimension number is the columns therefore the array has an overall shape of `(2, 3)`. In NumPy, it would be created as follows

```python
import numpy as np

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

print(arr)
```

the output is

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

Each `ndarray` has its own attributes and methods. The attributes of note are the `ndim`, `shape` and `size`. Just to name a few of the more commonly used methods are `reshape()`, `transpose()` and `flatten()`.

```python
# to get the number of dimensions
arr.ndim  # output: 2

# to get the shape of the array
arr.shape  # output: (2, 3)

# to get a count of all the elements in the array
arr.size  # output: 6
```

A feature for arrays is the **order**. The order is defined as the method of storing multidimensional arrays in linear storage (such as RAM). There are 2 types of orders **row major** and **column major**. Refer to figure 2 below.

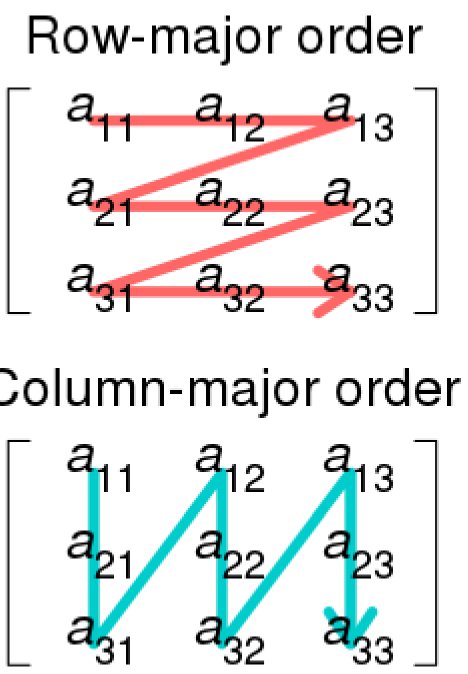
Row-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}
Column-major order
a_{11} a_{12} a_{13}
a_{21} a_{22} a_{23}
a_{31} a_{32} a_{33}

Figure 2: 2 ways of storing multidimensional arrays in computer memory.

Row major is read from left to right and column major is read from top to bottom. Array ordering matters when arrays are passed between different programs written in different languages and it also matters in data retrieval performance as modern CPUs process sequential data more efficiently than non-sequential data.

There are many ways to create arrays such as empty arrays, array fill with ones or zeros, identity arrays or even from existing data (which we will see in the later sections). The main points of array creation are as listed

- Can be created from Python lists or tuples using the NumPy `array()` function from the above example.
- If the size of the array is known but the elements are unknown, an array can still be created, NumPy has functions to create arrays with **initial placeholder content**.
- Creating sequences of numbers can be generated in a similar fashion to Python `range()` but it returns arrays instead of lists.
    - `arange(start, end, step)` returns **evenly** spaced values within a given interval. `step size` is required. The array returned varies in length as the step size influences the elements in the returned array.

    - `linspace(start, end, no. elements)` is functions similarly to `arange` but instead of `step size`, the user defines how many elements in the array do they want from the number range (inclusive of end range) given.
- Reshaping an array is done with the `reshape()`. It reshapes arrays to different dimensions. By default it is using **row major**.
- Flattening an array, that is to convert a multidimensional array to a **one dimensional** array. By default it is using **row major**.

13.2 Indexing and slicing

Array indexing in NumPy is similar to Python lists in the sense that they use integers for indexing and the colon (`:`) operator with integers for slicing (cutting up into smaller new arrays). In addition, NumPy arrays of boolean type (non zero), are able to be used in comparison operations without slicing.

- Slicing is done using the colon (`:`) operator, for instance `i:j:k` where `i` is the starting index, `j` is the stopping index and `k` step (k cannot equals 0)

```python
# basic slicing
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(x[1:7:2])
# Output: array([1, 3, 5])
```

- On top the indexing similarities with Python Lists, it has more advanced indexing features. This example creates a new array via cherry picking the elements from each row using each element from the column array. Imagine a table indexing system where the intersection between row and column indexes gives the desired element.

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

# [0, 1, 2] are the wanted row indexes
# [0, 1, 0] are the wanted column elements indexes w.r.t rows indexes
# the column element indexes position also determines the position
# of elements in the new array
x[[0, 1, 2], [0, 1, 0]]
# Output: array([1, 4, 5])
```

- Boolean array indexing is used when there is a need to pick elements from an array, based on some condition

```python
import numpy as np

x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
# picking all elements which are not np.nan values
x[~np.isnan(x)]
# Output: array([ 1.,  2.,  3.])
```

13.3 Data Types

NumPy supports a greater number of data types compared to Python as NumPy is written with C. It also uses the `dtype` object or any valid datatype that can be converted into a `dtype` object. When creating a new array, a `dtype` can be specified as follows

```
np.zeros(10, dtype='int16')
```

this creates an array of 10 zeros with an integer of type 16-bits or if it using the NumPy object

```
np.zeros(10, dtype=np.int16)
```

Some of the more common data types are listed in the table below. Note the size and precision of data types.

Data Type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)

Python data types are also acceptable. Structured datatypes are datatypes combinations for arrays as well.

```
import numpy as np

# combination datatype
# S50 - String, 50 characters
dt = np.dtype([('name', 'S50'), ('age', 'int16')])
a = np.array([('John', 20),('Hanna', 18),('Wayne', 60)], dtype = dt)
print(a)
```

the output is

```
[(b'John', 20) (b'Hanna', 18) (b'Wayne', 60)]
```

More on NumPy datatypes can be read from here.

13.4 Broadcasting

The general rules of array manipulation functions states that array inputs produces array outputs by performing **element-wise** operations on the inputs.

```
import numpy as np

# done element wise
a = np.array([1,2,3,4])
b = np.array([10,20,30,40])
c = a * b
print c
```

```
[10   40   90   160]
```

But what happens if the dimensions are not the same? Element-wise operations will not be possible in this case thus **broadcasting** is used where the smaller array is broadcasted to fit the size of the larger array. To be able to broadcast, the following rules must be satisfied

- The array with the smaller dimension have to be prepended (added to the beginning) with '1' to their shape.
- Sizes of the dimension of the output shape have to be the maximum sizes of the dimensions of the input shape.
- Input arrays can be used in calculation if the size in a particular dimension matches the output size or the value is exactly 1.
- If an input has a dimension of size 1, the first data element in that dimension is used for all calculations along that dimension.

For a set of arrays to be **broadcastable**, it has to produce a valid result from the rules above and one of the following is true

- Arrays have the exact same shape.
- Arrays can have the same number of dimensions and the length of each dimension is either a common length or 1
- Arrays with too few dimensions can have their shapes prepended with a dimension of length 1 so that the above property is satisfied.

```
import numpy as np

a = np.array([[0.0, 0.0, 0.0],
              [10.0, 10.0, 10.0],
              [20.0, 20.0, 20.0]])
b = np.array([1.0,2.0,3.0])

print(a+b)
```

the output is

```
[[ 1.  2.  3.]
 [11. 12. 13.]
 [21. 22. 23.]]
```

13.5 Mathematical Functions

The arithmetic operators (`+`, `-`, `*`, `/`) and numerous math functions from (trigonometric, hyperbolic, rounding, arithmetic, exponents, logarithms to complex numbers) are all performed **element-wise**. The full list can be found here. Note that by default, **radians** is used for all trigonometric functions.

```python
import numpy as np

a = [np.pi / 2, np.pi / 3, np.pi]

print("Original array:", a)
print()
# calculate the sine values in radians
print("Sine in radians:", np.sin(a))
# calculate the hyperbolic sine values
print("Hyperbolic Sine:", np.sinh(a))
# rounding to up to the nearest whole number by default
print("Rounding to the nearest whole number:", np.round(a))
# Natural logarithm
print("Natural logarithm:", np.log(a))
```

the output is

```
Original array: [1.5707963267948966, 1.0471975511965976, 3.141592653589793]

Sine in radians: [1.00000000e+00 8.66025404e-01 1.22464680e-16]
Hyperbolic Sine: [ 2.3012989   1.24936705 11.54873936]
Rounding to the nearest whole number: [2. 1. 3.]
Natural logarithm: [0.45158271 0.0461176  1.14472989]
```

Another very useful library that is you can use to with math, is the Random and Statistics library found here and here respectively. The Random library allows us to generate random numbers based on a pseudo-random number generator or via probability distributions and the Statistics library gives us functions such as `mean`, `std` for standard deviation, etc.

13.6 Matrix Library

Matrix library contains functions that returns **matrices** instead of `ndarray` objects. The library `from numpy import matlib` is required. The full list of functions can be found here. Arithmetic operations like `+`, `-`, and `*` are performed with the same matrix alignment rules from math. But if you are seeking element-wise operations, just stick to using `ndarray`s.

```python
from numpy import matlib

# creating a new 2 by 3 matrix
mat1 = matlib.mat('1,2,3;4,5,6')
print (mat1)
print()
# creating an identity matrix
id1 = matlib.identity(3, dtype = int)
print (id1)
```

the output is

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

[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

13.7 Linear Algebra Library

Linear Algebra consist of the algebraic functions on arrays such as dot products, inner product, eigenvectors, eigenvalues, determinants and more. The full list can be found here. Vector calculus functions are located in the normal `numpy` package and the rest can be found in the `linalg` library therefore we need to include `from numpy import linalg` in the workspace.

```python
import numpy as np
from numpy import linalg

a = np.array([[1,2],[3,4]])
b = np.array([[11,12],[13,14]])

# dot product
print(np.dot(a,b))

# determinant of an array
print(linalg.det(a))
```

the output is

```
[[37 40]
 [85 92]]

-2.0000000000000004
```

13.8 Input/Output with NumPy

`ndarray` objects can be saved and loaded from external files on disk. NumPy has numerous I/O functions but we will be focusing on 2 main types of I/O functions

- `load()` and `save()` for NumPy binary files (they have the `.npy` extension)
- `loadtxt()` and `savetxt()` for normal text files. Note that saving to text files is **only** for arrays up to **2 dimensions** for higher dimensions, use the NumPy binary files.

```python
import numpy as np

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

# save as binary file
np.save('outfile',a)

# load from binary file
b = np.load('outfile.npy')
print(b)
```

the output is

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

13.9 Matplotlib with NumPy

Plotting using NumPy data can be done through `ndarrays` or Python mutable data structures (lists, tuples and dictionaries). Just make sure that the both the values for x and y axis are of the same shape.

```python
# x and y have ndarrays of size 50
data_d = {'x': np.arange(50),
          'y': np.random.randint(0, 50, 50)}
# scatter plot
scatter('x', 'y', data=data_d)
```

In the above example, `data_d` is a Python Dictionary with NumPy arrays as values. The dictionary keys are used as labels therefore they can be used in replacement to normal variables to fill the `plot` function's arguments.

13.10References

1. Python Numpy, https://www.geeksforgeeks.org/
2. Row- and column-major order, https://en.wikipedia.org/wiki/Row-_and_column-major_order
3. NumPy Reference, https://numpy.org/doc/1.18/reference/index.html
4. Lubanovic, 2019, 'NumPy, Chapter 22. Py Sci' in Introducing Python, 2nd Edition, O`Reilly Media, Inc.