NumPy

Gabriel S. Cabrera

August 23, 2018

Contents

1 Motivation **2**

1.1 Basic Arithmetic 3

1.2 Miscellaneous Operations 3

1.3 More Vector Operations 4

1.4 Matrix Operations 4

2 Introduction **5**

3 Converting Lists to NumPy Arrays **6**

3.1 One Dimension 6

3.2 Two Dimensions 7

3.3 Column Vectors 8

3.4 Higher Dimensions 9

4 Array Properties and Usage **12**

4.1 Array Shape 12

4.2 Getting Elements 13

4.2.1 1-D Arrays 13

4.2.2 2-D Arrays 13

4.2.3 3-D Arrays 14

4.2.4 Higher Dimensional Arrays 14

4.3 Getting Sub-Arrays 15

4.3.1 1-D Arrays 15

4.3.2 2-D Arrays 16

4.3.3 3-D Arrays 16

4.4 Setting Elements and Sub-Arrays 17

4.4.1 1-D Arrays 17

4.4.2 2-D Arrays . . . 18
4.4.3 3-D Arrays . . . 19
4.5 Axis . . . 20
4.6 Broadcasting . . . 21
4.7 Datatypes . . . 21

5 Logic **22**
5.1 Element-Wise Comparators . . . 22
5.2 Element-Wise Boolean Operators . . . 23
5.3 Other Comparators . . . 23

6 Arithmetic **24**

7 Useful Functions **24**
7.1 absolute . . . 24
7.2 amax . . . 25
7.3 amin . . . 25
7.4 arange . . . 25
7.5 argmax . . . 26
7.6 argmin . . . 26
7.7 diff . . . 27
7.8 linspace . . . 27
7.9 maximum . . . 28
7.10 meshgrid . . . 28
7.11 minimum . . . 29
7.12 ones . . . 29
7.13 ones_like . . . 29
7.14 prod . . . 30
7.15 sum . . . 30
7.16 zeros . . . 30
7.17 zeros_like . . . 31

1 Motivation

This section can be considered somewhat independently of the rest of this chapter; for the experienced but rusty, it is intended as a quick-reference. For the inexperienced it can be seen as a motivator – it highlights many of the key concepts one should understand about vectorization, and is a showcase of what a novice can accomplish once they've read through this chapter.

1.1 Basic Arithmetic

Addition, subtraction, multiplication, division, and exponentiation is possible between two equal-sized arrays *or* between an array and a `float`/`int`:

Input

```
import numpy as np

a = np.array([2, 3, 6, 8])
b = np.array([8, 6, 2, 4])

print(a + b)
print(a + 1)

print(b - a)
print(a - 1)

print(a * b)
print(a * 2)

print(a / b)
print(a / 2)

print(a ** b)
print(a ** 2)
```

Output

```
[10  9  8 12]
[3 4 7 9]
[ 6  3 -4 -4]
[1 2 5 7]
[16 18 12 32]
[ 4  6 12 16]
[0.25 0.5  3.   2.  ]
[1.  1.5 3.  4. ]
[ 256  729   36 4096]
[ 4  9 36 64]
```

1.2 Miscellaneous Operations

Numpy has countless other available operators, such as the square root (`sqrt`), natural exponential function (`exp`), the natural logarithm (`log`), and the base-10 logarithm (`log10`).

Input

```
import numpy as np

a = np.array([1, 4, 9, 16])

print(np.sqrt(a))
print(np.exp(a))
print(np.log(a))
print(np.log10(a))
```

Output

```
[1. 2. 3. 4.]
[2.71828183e+00 5.45981500e+01 8.10308393e+03 8.88611052e+06]
[0.         1.38629436 2.19722458 2.77258872]
[0.         0.60205999 0.95424251 1.20411998]
```

1.3 More Vector Operations

Here we show examples of some very powerful vector commands.

Input

```
1  import numpy as np
2
3  a = np.array([1, 2, 3, 4])
4
5  print(np.sum(a**2))
6  print(np.amin(a))
7  print(np.amax(a))
8  a[np.where(a == 3)] = 7
9  print(a)
10 x = np.where(a > 2)
11 print(x)
12 print(a[x])
```

Output

```
30
1
4
[1 2 7 4]
(array([2, 3]),)
[7 4]
```

1.4 Matrix Operations

Matrices can be handled in the same way.

Input

```
1  import numpy as np
2
3  a = np.array([[1, 2], [6, 7]])
4  print(a)
5  print(a**2)
6  print(a[0:2,1])
7  print(np.sum(a))
8  print(np.transpose(a))
```

Output

```
[[1 2]
 [6 7]]

[[ 1  4]
 [36 49]]

[2 7]

16

[[1 6]
 [2 7]]
```

2 Introduction

When processing data, one must choose between a straightforward, simple, but slower language (such as Python or Lua) or a more complex but faster language (such as C++ or Fortran) – simplicity (usually) results in less efficient usage of a computer's resources.

Fortunately there is a middle ground for Python: the NumPy module. Generally speaking, working with sets of numbers in Python requires us to iterate through lists, a slow process relative to looping in lower-level languages such as C++, but NumPy allows us to *vectorize* our lists such that they can be processed as single units allowing for more elegant code and much faster processing.

There is a downside to NumPy however: vectorized arrays are static in length – once created, an array's length is unchangeable. This means that we lose both the ability to append new elements to an array, as well as the ability to delete existing ones. In addition to this, NumPy is unable to improve performance for the integration of second-order differential equations[1].

Example Let's say we have two sets of numbers of equal length – if we wish to add them to each other element-wise, Python normally requires us to loop through them as follows:

```
1 a = [0,1,2,3,4,5,6,7,8,9]
2 b = [9,8,7,6,5,4,3,2,1,0]
3 
4 c = []
5 for i,j in zip(a, b):
6     c.append(i+j)
```

To perform the same operation as above in NumPy is a lot simpler, as shown below:

```
1 import numpy as np
2 
3 a = np.array([0,1,2,3,4,5,6,7,8,9])
4 b = np.array([9,8,7,6,5,4,3,2,1,0])
5 
6 c = a + b
```

In this case, NumPy is slightly slower than standard Python[2], but we quickly see that it is advantageous as we make the lists longer and longer – in fact, Figure 1 shows that NumPy is on average

[1]To improve performance in such cases, take a look at the Numba module.
[2]Though only by a fraction of a second

12 times more efficient than using a `Python` loop when using arrays of sufficient length. This efficiency only gets better as we deal with more complex operations!

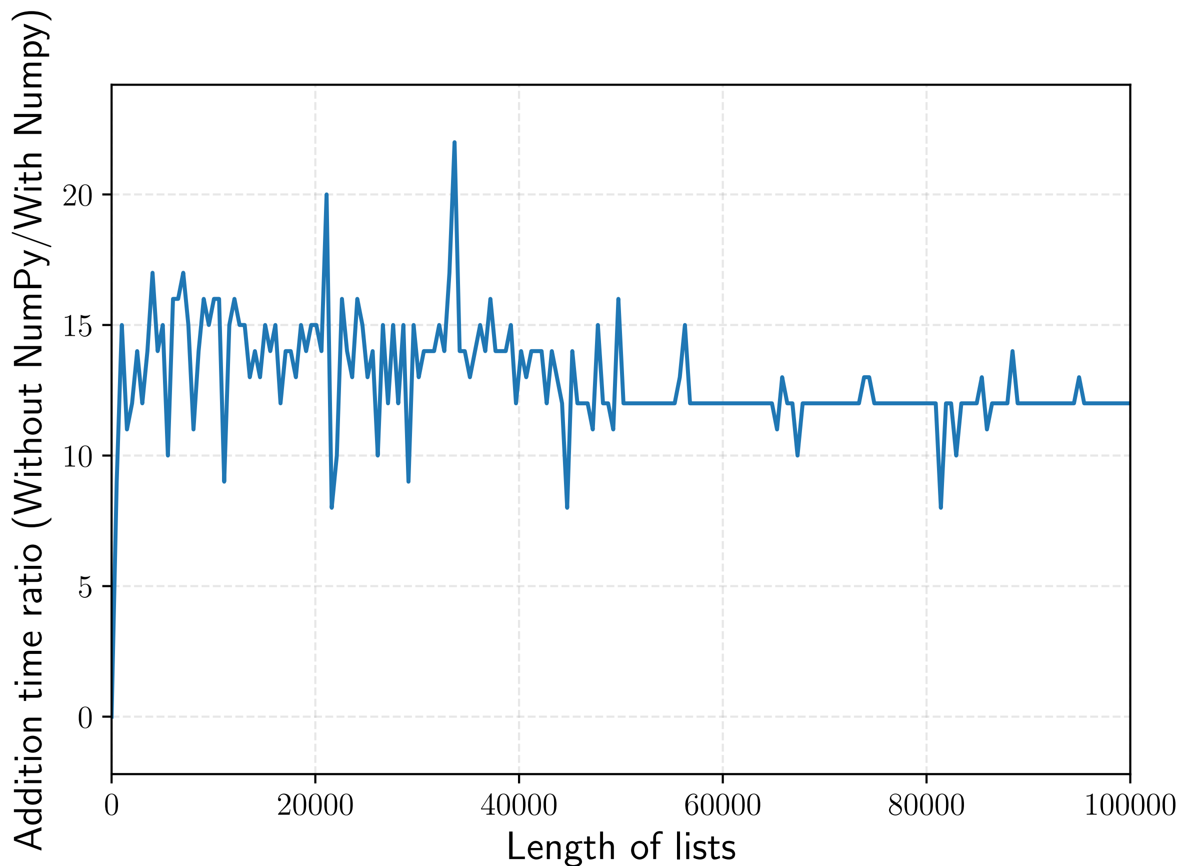


Figure 1: How `NumPy` can improve performance in array addition based on the lengths of given arrays

3 Converting Lists to NumPy Arrays

In `NumPy`, it is possible to generate arrays of all shapes, sizes, and dimensions. Everything from vectors to matrices to tensors can be recreated as arrays, however we should start by understanding 1-D arrays before moving on to higher dimensions.

3.1 One Dimension

To understand 1-D `array` objects, we will begin by defining a generalized vector:

Let $\vec{v} \in \mathbb{R}^n$ for $n \in \mathbb{N}$; this means that we have a vector $\vec{v}$ of length n, where n is any whole-number greater than zero. Our vector can also be represented by the following notation:

$$\vec{v} = \begin{bmatrix} v_1 & v_2 & \cdots & v_n \end{bmatrix} \tag{1}$$

In `NumPy`, 1-D arrays are *row-vectors* as opposed to *column-vectors*; this will become relevant later when we look at 2-D arrays.

Example Vectors can take many forms, such as $\vec{a} = \begin{bmatrix} 1 & -2 & 5 \end{bmatrix}$ or $\vec{b} = \begin{bmatrix} 0.01 & 0.5 \end{bmatrix}$ or $\vec{c} = \begin{bmatrix} 5 & 2.5 & 81 & 32 & -33.5 \end{bmatrix}$.

We can recreate these vectors in `array` form, as shown below:

```
1 import numpy as np
2
3 a = np.array([1, -2, 5])
4 b = np.array([0.01, 0.5])
5 c = np.array([5, 2.5, 81, 32, -33.5])
```

3.2 Two Dimensions

As with `lists` and `tuples`, we can create 2-D arrays with `NumPy` – alternatively, we can also create `matrix`-objects[3], though this will not be our focus.

Once again, let us take a look at a natural mathematical analogy: matrices. Let's describe them in three different ways:

Formally Let $A \in \mathbb{R}^{m \times n}$ be an $m \times n$ matrix of elements $a_{i,j} \in \mathbb{R}$ with $m, n \in \mathbb{N}$, $m \geq i \in \mathbb{N}$, and $n \geq j \in \mathbb{N}$.

Intuitively We have a matrix A consisting of real numbers with m rows and n columns.

Visually

$$\begin{bmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \cdots & a_{m,n} \end{bmatrix} \tag{2}$$

In `NumPy`, 2-D arrays are created in much the same way as 1-D arrays:

Example 1 Let's say we have the 3×3 identity matrix:

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3}$$

We can use `NumPy` to create a corresponding 2-D array, we must simply separate I_3 row-by-row:

[3] For an overview of the differences between `array` and `matrix` objects, see: https://www.scipy.org/scipylib/faq.html#what-is-the-difference-between-matrices-and-arrays

```python
import numpy as np

I_3 = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]])
```

Where printing `I_3` gives us the following output:

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

Example 2 We are given the following 6×4 matrix:

$$\begin{bmatrix} 3 & 3 & 1 & 6 \\ 7 & 4 & 0 & -3 \\ 0 & -2 & 9 & 7 \\ -2 & 3 & 0 & 1 \\ 5 & 7 & 1 & 1 \\ 8 & 8 & 1 & -8 \end{bmatrix} \tag{4}$$

We can once again separate our matrix into row-vectors to create a 2-D array:

```python
import numpy as np

M = np.array([[3, 3, 1, 6], [7, 4, 0, -3], [0, -2, 9, 7], [-2, 3, 0, 1], [5, 7, 1, ↩
    1], [8, 8, 1, -8]])
```

Printing `M` gives us the output below:

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

3.3 Column Vectors

Recall from Section 3.1 that 1-D `NumPy` arrays are row-vectors; this means that if we wish to create a column vector we must set up a 2-D array with single-element arrays.

Opting to use column vectors by default may come in handy when programming for linear algebra[4], since it may reduce the amount of effort required to perform operations between vectors and matrices.

Example We are given the unit vectors for $\mathbb{R}^3$ as follows:

[4]If you are intending to program with a heavy focus on linear algebra, it may be worthwhile to look into `NumPy`'s `matrix` object.

$$\hat{x} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \hat{y} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \hat{z} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} \tag{5}$$

To model these vectors using `NumPy`, we must create a set of 2-D arrays each consisting of 3 nested arrays, each of which only contain a single element. A functioning script is shown below:

```
1 import numpy as np
2
3 x = np.array([[1], [0], [0]])
4 y = np.array([[0], [1], [0]])
5 z = np.array([[0], [0], [1]])
```

Printing `x`, `y`, and `z` gives us the following output:

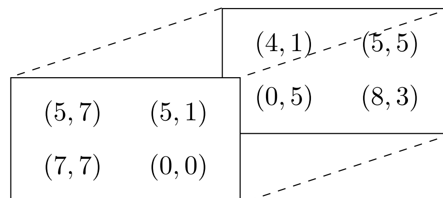
```
[[1]
 [0]
 [0]]

[[0]
 [1]
 [0]]

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

3.4 Higher Dimensions

We've now described the process used to generate 1-D and 2-D arrays, so what's next? Specifically, `NumPy` limits us to a maximum[5] dimension of 32, though this should not be an issue in most situations.

There are many uses for multidimensional arrays such as tensor[6] operations, dealing with partial differential equations, or modeling a system of objects in a field. Let us look at a few examples:

Example 1 Let's model a 4th-order multidimensional array, an object that can be visualized as a 4-D matrix. Here we have an example of a $2 \times 2 \times 2 \times 2$ array:



To create this using `NumPy` we must simply nest our lists in the desired hierarchy, making sure to use the row-vectors once more:

[5] Attempting to initialize an array with more than 32 dimensions will raise an exception – `ValueError: sequence too large; must be smaller than 32`

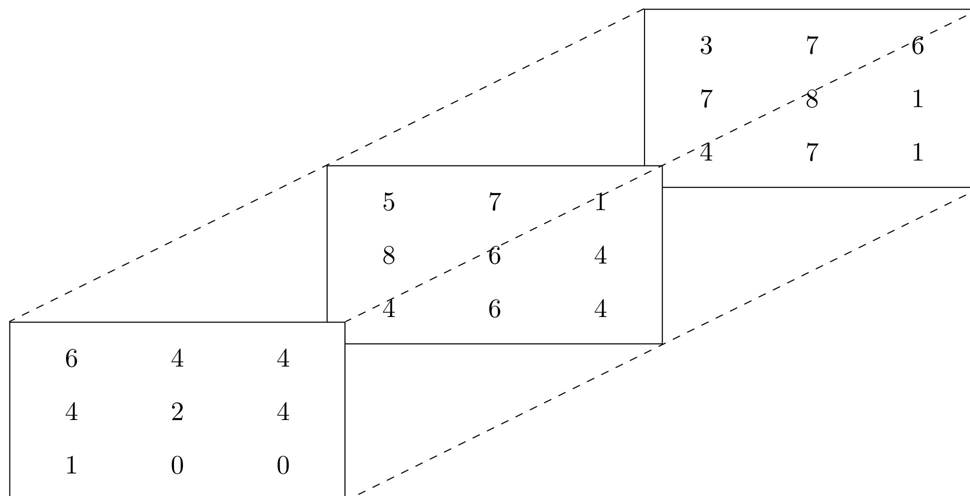
[6] Information on Tensors available at Wikipedia: https://en.wikipedia.org/wiki/Tensor

```
1 import numpy as np
2
3 M = np.array([[[[5, 7], [5, 1]], [[7, 7], [0, 0]]],
4 [[[4, 1], [5, 5]], [[0, 5], [8, 3]]]])
```

Printing the M array gives us the following output:

```
[[[[5 7]
   [5 1]]

  [[7 7]
   [0 0]]]


 [[[4 1]
   [5 5]]

  [[0 5]
   [8 3]]]]
```

Example 3 Let's now create a 3rd-order multidimensional array, which is similar to a 3-D matrix. Let us take a look at a $3 \times 3 \times 3$ array:



To create this using NumPy, we must once again nest our lists, while being careful to maintain the correct dimensions for our array:

```
1 import numpy as np
2
3 M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
     [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])
```

Printing the M array gives us the following output:

```
[[[6 4 4]
  [4 2 4]
  [1 0 0]]
```

```
[[5 7 1]
 [8 6 4]
 [4 6 4]]

[[3 7 6]
 [7 8 1]
 [4 7 1]]]
```

4 Array Properties and Usage

4.1 Array Shape

In NumPy there is a standardized system that is used to describe an array's dimensions. In mathematical notation, we would say that the following is a 3×2 matrix:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} \tag{6}$$

This can be reproduced in NumPy via the following:

```
1 import numpy as np
2
3 M = np.array([[1, 2], [3, 4], [5, 6]])
```

Printing M gives us the output below:

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

In NumPy, we would say that the dimensions of the array equivalent to Equation 6 is (3,2); this is called the array's *shape*[7], which can be found by accessing the array's shape attribute:

```
1 import numpy as np
2
3 M = np.array([[1, 2], [3, 4], [5, 6]])
4 print(M.shape)
```

This gives us the following output:

```
(3, 2)
```

So in short, the array shape tells us the "depth" of our object in each of its directions, with the order of these depths shown in the array's shape tuple. Programmatically speaking, the order of these values is determined by the nesting order chosen during the array's initialization.

[7]More information on the shape attribute can be found on SciPy's website: https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html

4.2 Getting Elements

Extracting and modifying individual elements from a `NumPy` array is relatively straightforward, as the required syntax is equivalent to the syntax used for lists and tuples, but it is also important to know how to extract or modify entire columns, rows, or planes in an array. In fact, we can extract any homogenous[8] sub-array from `NumPy` arrays.

Let's begin with reviewing how to get individual elements. Here are a few quick examples as a reminder:

4.2.1 1-D Arrays

Let's take a look at the following vector:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{7}$$

Printing 5:

```
1 import numpy as np
2
3 a = np.array([4, 2, 7, 5, 3, 1])
4 print(a[3])
```

4.2.2 2-D Arrays

We will be examining the following matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{8}$$

Single Elements Printing 6:

```
1 import numpy as np
2
3 A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
4 print(A[2][1])
```

Subarrays Printing `[5 3 8]`:

```
1 import numpy as np
2
3 A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
4 print(A[0])
```

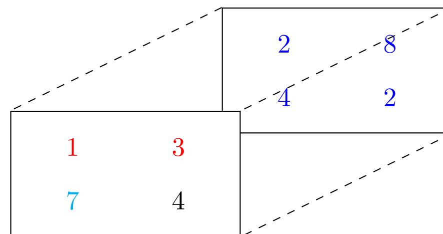
For an $m \times n$ array `A`, the following guide will give us an element's index:

[8] Homogeneity in the sense that all its rows are of equal size in their respective depths.

$$
\begin{bmatrix}
\texttt{A[0][0]} & \texttt{A[0][1]} & \cdots & \texttt{A[0][n-1]} \\
\texttt{A[1][0]} & \texttt{A[1][1]} & \cdots & \texttt{A[1][n-1]} \\
\vdots & \vdots & \ddots & \vdots \\
\texttt{A[m-1][0]} & \texttt{A[m-1][1]} & \cdots & \texttt{A[m-1][n-1]}
\end{bmatrix} \tag{9}
$$

4.2.3 3-D Arrays

We will be working with the following 3-D matrix:



Single Elements Printing 7:

```
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[0][1][0])
```

1-D Subarrays Printing `[1 3]`:

```
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[0][0])
```

2-D Subarrays Printing `[[2 8] [4 2]]`:

```
import numpy as np

A = np.array([[[1, 3], [7, 4]], [[2, 8], [4, 2]]])
print(A[1])
```

4.2.4 Higher Dimensional Arrays

The principles introduced for 1-D, 2-D, and 3-D arrays will carry over to higher dimensions, the trick is to really know the shape of your array before implementing your code. In other words, printing your array's `shape` (as introduced in Section 4.1) can be a useful aid in determining exactly where your desired values are located.

4.3 Getting Sub-Arrays

Getting an array's elements is relatively straightforward, but let's say we wish to extract the second column of a matrix – this sort of task can be accomplished inefficiently and in a convoluted manner by creating an empty array and appending it with the second element of each row while looping through it, but we are interested in efficiency and elegance.

`NumPy` gives us such an option via the use of `slice` objects, whereby a user can define a range of values to extract from an array. Slice objects in this case are defined as follows: `[start:stop:step]`, where each of these values must be of type `int`. In addition to this, we have that:

- If the slice is defined as `[start:stop]` the `step` is set equal to `1` by default.
- It is possible to leave either `start` or `stop` or *both* blank; in such a case, `start` would be assigned a default value of `0`, while `stop` would include all subsequent elements past `start`.
- We can use the `step` option on an entire array with `[::step]`. Using a `slice` such as `[::2]` would reference an array's every other element.
- The `step` can be negative! This simply reverses the order of iteration through the array and then applies the chosen `step` – be sure to make `start` > `stop` however.
- Using a slice such as `[::-step]` will reverse the entire array, and iterate through it with the given `step`.
- For multidimensional arrays we can use the same syntax, but with a comma dividing individual "blocks". For example, a 2-D array could have a slice formatted in such a way that `[start1:stop1:step1:, start2:stop2:step2]`, where the depth of the array being referenced increases with each added block.

As previously, this syntax is best taught via example, so we will show how to extract values in 1-D, 2-D, and 3-D arrays.

4.3.1 1-D Arrays

We are given the following vector:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{10}$$

Printing `[2 7 5]`:

```
1 import numpy as np
2
3 a = np.array([4, 2, 7, 5, 3, 1])
4 print(a[1:4])
```

Note Take a close look at our subarray `a[1:4]`, and note that the first boundary integer 1 is the same as the index of `2` within our original array `a`. Now, observe that our second boundary integer 4 is greater than the index of `5` within our original array `a`.

In slicing syntax, the first boundary integer should therefore correspond to the lower index of our sub-array, while the second boundary integer should be the higher index of the sub-array +1.

4.3.2 2-D Arrays

We have the following matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{11}$$

Columns Printing `[3 3 6]`:

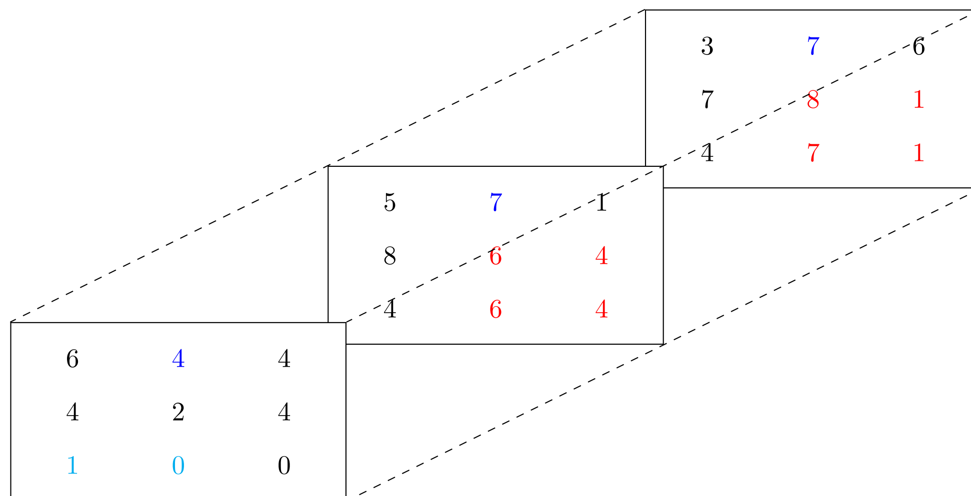
```
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[:,1])
```

Subarrays Printing `[8 5]`:

```
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
print(A[0:2,2])
```

4.3.3 3-D Arrays

Things start to get complex when dealing with 3-D arrays, since each added dimension vastly increases the slicing possibilities[9]. We will be working with the following 3-D matrix:



Rows Printing `[4 7 7]`:

[9]Things gets even more complex as we hit 4-D arrays, since these are far more difficult to visualize.

```
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[:,0,1])
```

Cubes Printing `[[[6 4] [6 4]] [[8 1] [7 1]]]`:

```
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[1:,1:,1:])
```

More Rows Printing `[1 0]`:

```
import numpy as np

M = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4],
[4, 6, 4]], [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])

print(M[0,2,:-1])
```

4.4 Setting Elements and Sub-Arrays

This section will be brief, as it builds upon Sections 4.2 and 4.3.

Since arrays in `NumPy` have a fixed size upon initialization, we must rely on changing elements directly rather than deletion and appending with `del` and `append`, respectively. In the aforementioned sections, we described how to access elements and sub-arrays via the usage of indices and `slice` objects – the theory here is the same, and only requires a slight change in syntax.

In short, we define an index/slice "`i`" of an array "`a`" and set it to a value "`v`" whose shape must be broadcastable[10] to that of the selected slice[11] with the syntax `a[i] = v`.

We will use a few examples to illustrate this more clearly.

4.4.1 1-D Arrays

Example 1 We are given the following array:

$$\begin{bmatrix} 4 & 2 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{12}$$

We can change 2 into 0 with the following:

[10] Broadcasting is discussed later in Section 4.6. For now, think of a broadcastable shape as one that matches the shape of the other, but in any desired orientation.

[11] Of course, if we are replacing an individual number, then our value should also be a number, not an array.

```
import numpy as np

a = np.array([4, 2, 7, 5, 3, 1])
a[1] = 0
```

Printing `a` gives us the output below:

```
[4 0 7 5 3 1]
```

Example 2 Continuing from the previous example, we now have the following array:

$$\begin{bmatrix} 4 & 0 & 7 & 5 & 3 & 1 \end{bmatrix} \tag{13}$$

Let's say we wish to replace 7, 5, and 3 with 1, 3, and 5. This can be accomplished by first creating an array matching the three new values: `[1 3 5]`

```
import numpy as np

a = np.array([4, 0, 7, 5, 3, 1])
b = np.array([1, 3, 5])
a[2:5] = b
```

Printing `a` gives us the output below:

```
[4 0 1 3 5 1]
```

4.4.2 2-D Arrays

We are given a matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{14}$$

We can swap out the column containing `[3 3 6]` for another array[12] `[1 2 3]` with the following script.

```
import numpy as np

A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
b = np.array([1, 2, 3])
A[:,1] = b
```

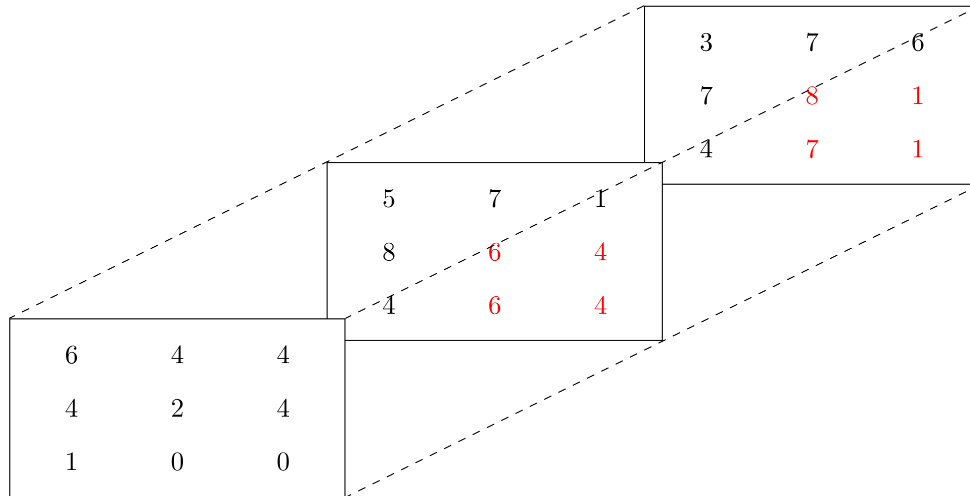
Printing `A` then gives us the following output:

[12]Even though we are technically swapping a column for a row, this still works due to the concepts we will bring up in Section 4.6.

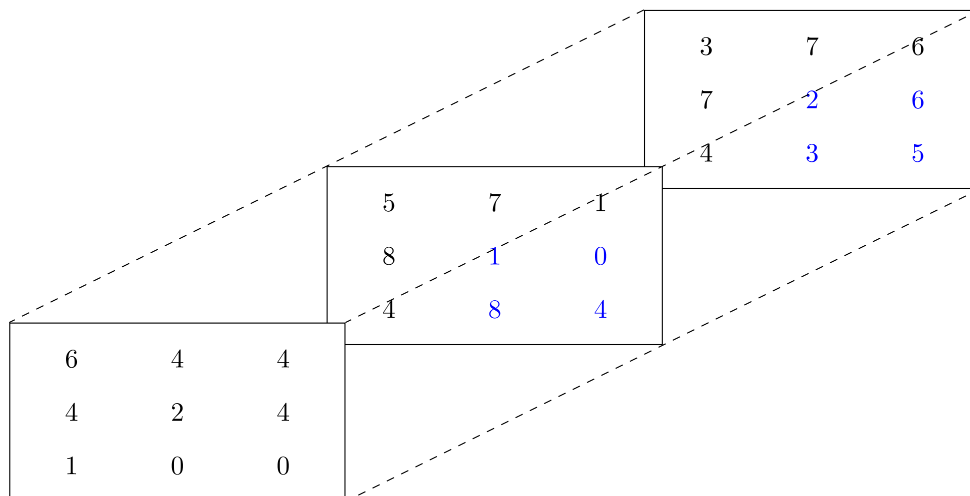
```
[[5 1 8]
 [4 2 5]
 [3 3 1]]
```

4.4.3 3-D Arrays

We are given a 3-D matrix:



We are interested in switching out our matrix' back right corner, creating a new matrix below:



To make these changes efficiently, we must create a $2 \times 2 \times 2$ array and set up the `slice` for the red elements' positions to this new array: `[1:,1:,1:]`.

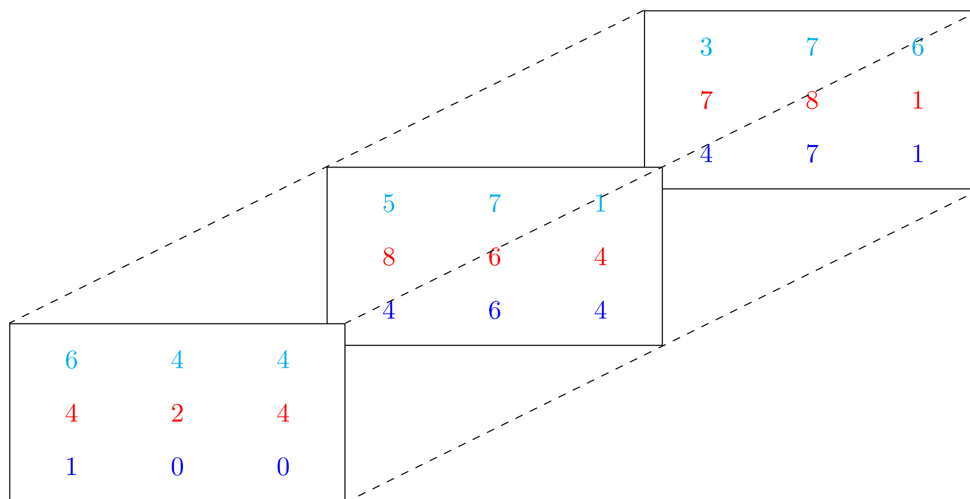
We have a script to accomplish this for us:

```
1 import numpy as np
2
3 A = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
      [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])
4
5 b = np.array([[[1, 0], [8, 4]], [[2, 6], [3, 5]]])
6
7 A[1:,1:,1:] = b
```

4.5 Axis

When dealing with multidimensional arrays, there are situations where a `NumPy` function needs to know in which direction it should operate. This is where the array `axis` comes in – the concept is rather simple once we understand how `shape` works, since the `axis` of an array is simply the term defining whether we are referring to a row, column, or other direction when dealing with 3+ dimensions. In short, the axis is simply the index given for an array direction as defined by the order in its `shape` attribute.

Example We are given the following 3-D array:



Each color represents one of the elements grouped over our array's 1st axis, this is because our first axis is perpendicular to the planes we've color coded, thus defining the selected axis. Orthogonality can always be used to determine which axis we are working with, though this can be difficult to picture in higher dimensions.

In 3-D, we can picture that an operation will occur between corresponding members of each `axis` in the plane defined by the `axis` direction. When dealing with N dimensions, the operation will occur between all parallel (N-1) dimensional sub-arrays orthogonal to the given `axis`.

We can take the sum over the color coded 1st axis with the script below:

```
1 import numpy as np
2
```

```
A = np.array([[[6, 4, 4], [4, 2, 4], [1, 0, 0]], [[5, 7, 1], [8, 6, 4], [4, 6, 4]], ↩
    [[3, 7, 6], [7, 8, 1], [4, 7, 1]]])
print(np.sum(A, axis = 1))
```

This will add each colored group to each other, giving us the following output:

```
[[11  6  8]
 [17 19  9]
 [14 22  8]]
```

4.6 Broadcasting

In Section 4.4.2, we were able to swap out a column in a matrix with a simple row vector. But how is this possible if the arrays have different shapes? It's because NumPy uses a clever compatibility system called **broadcasting** that allows operations to be performed between arrays of similar yet different shapes or between scalars and arrays. This is why we can multiply scalars by matrices in NumPy, because the scalar's value is *broadcasted* to the shape of the array, allowing for element-wise multiplication to occur.

It is highly recommended that new users read the documentation provided by SciPy on the matter: https://docs.scipy.org/doc/numpy-1.13.0/user/basics.broadcasting.html

4.7 Datatypes

To improve a program's runtime, NumPy requires that array elements have a static type – by default, an array will try and determine what type it should contain upon initialization, when it detects the types in its input list/tuple. We can however force an array to take on a specific type, a process that is often overlooked at critical moments.

Example Let's say we wish to create a linearly spaced array with the linspace function referenced in Section 7.8, then redefine an element by adding a complex scalar to it.

```
import numpy as np

a = np.linspace(1, 10, 10)
a[5] = a[5] + 1j
print(a)
```

Attempting to run the above script will result in a warning and an undesirable output, as shown below:

```
script.py:4: ComplexWarning: Casting complex values to real discards the imaginary part
  a[5] = a[5] + 1j
[ 1.  2.  3.  4.  5.  6.  7.  8.  9. 10.]
```

If we wish to be sure our array can handle complex variables, we need to set its dtype[13] to complex upon initialization. So to fix the above script, we implement the following:

[13] A shortening of *datatype*

```
import numpy as np

a = np.linspace(1, 10, 10, dtype = complex)
a[5] = a[5] + 1j
print(a)
```

And now we get the output we desire:

```
[ 1.+0.j  2.+0.j  3.+0.j  4.+0.j  5.+0.j  6.+1.j  7.+0.j  8.+0.j  9.+0.j
 10.+0.j]
```

The lesson to learn here is that we must always take care to create arrays that can handle any datatypes that we intend to insert into them as elements.

5 Logic

When dealing with simple objects in `Python` such as `bool`s, `float`s, or `int`s, we often take for granted the simplicity with which we can use comparators such as `==`, or `>=`. This is something that takes getting used to in `NumPy` (and when using arrays in general), since we now have different ways in which to compare arrays.

5.1 Element-Wise Comparators

The comparators in this subsection are used by calling the desired function directly from **NumPy** and passing two equally shaped arrays as arguments. Its output will then be a boolean array of equal shape, with each pair of elements evaluated in their respective indices.

Example We are given two arrays `a` and `b`; to check which of their elements are equal element-wise, we can use the `equal` function:

```
import numpy as np

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

Printing `c` gives us the following output:

```
[[ True  True False]
 [ True  True False]]
```

Table of Comparators Since we don't have the luxury of using all our standard comparators, we can use Table 1 as a guide. We can refer to it when we wish to make element-wise comparisons between two arrays `a` and `b`.

Table 1: Element-wise comparators for two equal-shaped NumPy arrays a and b

Standard Python	NumPy Arrays
a == b	equal(a, b)
a != b	not_equal(a, b)
a > b	greater(a, b)
a < b	less(a, b)
a >= b	greater_equal(a, b)
a <= b	less_equal(a, b)

5.2 Element-Wise Boolean Operators

If we intend to set up more complex logical statements, we will need to have access to logical operations. Let us once again begin with an example:

Example Let's negate[14] an array a with the logical_not function.

```
import numpy as np

a = np.array([[True, True], [False, False]])
b = np.logical_not(a)
```

Printing b gives us the output below:

```
[[False False]
 [ True  True]]
```

Table of Operators This section can be referred to when looking for an element-wise logical operation.

Table 2: Element-wise logical operations for two equal-shaped NumPy arrays a and b

Standard Python	NumPy Arrays
not a	logical_not(a)
a and b	logical_and(a, b)
a or b	logical_or(a, b)
a ^ b	logical_xor(a, b)

5.3 Other Comparators

There are a few other comparators that may come in handy:

[14] Meaning, we will use the element-wise equivalent to the not operation.

Table 3: Element-wise logical operations for two equal-shaped NumPy arrays a and b

NumPy Syntax	Description
array_equal(a, b)	True if a and b have the same shape and elements.
array_equiv(a, b)	True if a and b are broadcastable with the same elements.

6 Arithmetic

NumPy's ability to vectorize arithmetic is arguably its most useful feature – it allows us to run operations such as addition, subtraction, multiplication, division, and exponentiation between any two broadcastable arrays in a single step. We can also perform other operations on individual arrays, such as taking their square root with sqrt, or setting it up as an exponent in exp.

Table of Operations Here is an overview of operations that can be taken between two broadcastable arrays a and b.

Table 4: Arithmetic operations for two broadcastable NumPy arrays a and b

Operation	Simple NumPy Syntax	NumPy Function
Addition	a + b	np.add(a, b)
Subtraction	a - b	np.subtract(a, b)
Multiplication	a * b	np.multiply(a, b)
Division	a / b	np.divide(a, b)
Floor Division	a // b	np.floor_divide(a, b)
Modulus	a % b	np.mod(a, b)
Exponentiation	a ** b	np.power(a, b)

Other useful mathematical functions can be found in SciPy's documentation: https://docs.scipy.org/doc/numpy-1.14.0/reference/routines.math.html

7 Useful Functions

Here is an overview of some useful functions a NumPy user should be familiar with; it is recommended that one read the descriptions for any unfamiliar ones, as they may come unexpectedly handy.

7.1 absolute

Takes the absolute value of an array element-wise – in cases with complex elements, it will find their real-valued absolute value.

So given the following:

```
1 import numpy as np
2
```

```
a = np.array([1, -3, 5, 3+4j, -8, 5+12j])
print(np.absolute(a))
```

We get this resulting output:

```
[ 1.   3.   5.   5.   8. 13.]
```

7.2 amax

Given an array and an axis, finds the element-wise maximum values.

Let's say we wish to find the maxima for `axis = 2`, then we can use the following:

```
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.amax(a, axis = 2))
```

Giving us our output:

```
[[2 4]
 [6 8]]
```

7.3 amin

Given an array and an axis, finds the element-wise minimum values.

Let's say we wish to find the minima for `axis = 2`, then we can use the following:

```
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.amin(a, axis = 2))
```

Giving us our output:

```
[[1 3]
 [5 7]]
```

7.4 arange

Creates an array of linearly-spaced values from `x0` up to (but not necessarily including[15]) `x1`, with a step of `dx` between them.

To create an array of values between -25 and 25 with a step of 1.5, we can use the following:

[15] `x1` may not always be included in the resulting array, since `x1` may not be evenly divisible by `dx`.

```python
import numpy as np

x0 = -25
x1 = 25
dx = 1.5

a = np.arange(x0, x1, dx)
```

Printing `a` give us the following output:

```
[-25.   -23.5 -22.   -20.5 -19.   -17.5 -16.   -14.5 -13.   -11.5 -10.    -8.5
  -7.    -5.5  -4.    -2.5  -1.     0.5   2.     3.5   5.     6.5   8.     9.5
  11.    12.5  14.    15.5  17.    18.5  20.    21.5  23.    24.5]
```

As in the case with `linspace`, we are able to use non-integer steps in `arange` generated arrays.

7.5 argmax

Similar to `amax`, with one crucial difference – rather than directly returning the maxima, returns the indices for the maxima.

Let's say we wish to find the maximum indices for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.argmax(a, axis = 2))
```

Giving us our output:

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

7.6 argmin

Similar to `amin`, with one crucial difference – rather than directly returning the minima, returns the indices for the minima.

Let's say we wish to find the minimum indices for `axis = 2`, then we can use the following:

```python
import numpy as np

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(np.argmin(a, axis = 2))
```

Giving us our output:

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

7.7 diff

Given an array, takes the difference between each sub-array and its preceeding sub-array for a given `axis`. This is best explained visually, so let's take a look at the following 2-D matrix:

$$\begin{bmatrix} 5 & 3 & 8 \\ 4 & 3 & 5 \\ 3 & 6 & 1 \end{bmatrix} \tag{15}$$

Let's say we wish to find the difference between each colored column, how can we accomplish this? Using Section 4.5 as a guide, we must apply our function on the axis orthogonal to the groups we wish to operate on – in this case, we see that the columns are separated along the 0th axis, implying that we should use `diff` on the 1st axis:

```
1 import numpy as np
2
3 A = np.array([[5, 3, 8], [4, 3, 5], [3, 6, 1]])
4 print(np.diff(A, axis = 1))
```

This gives us the following output:

```
[[-2  5]
 [-1  2]
 [ 3 -5]]
```

We can also infer by the signs of each element above that `diff` performs its operation such that we subtract the elements at lower indices from the elements in higher indices, in this case subtracting from right to left.

7.8 linspace

Creates an array of `N` linearly-spaced values between `x0` and `x1`.

To create an array of 101 values from -25 to 25, we can use the following:

```
1 import numpy as np
2
3 x0 = -25
4 x1 = 25
5 N = 101
6
7 a = np.linspace(x0, x1, N)
```

Printing `a` give us the following output:

```
[-25.  -24.5 -24.  -23.5 -23.  -22.5 -22.  -21.5 -21.  -20.5 -20.  -19.5
 -19.  -18.5 -18.  -17.5 -17.  -16.5 -16.  -15.5 -15.  -14.5 -14.  -13.5
 -13.  -12.5 -12.  -11.5 -11.  -10.5 -10.   -9.5  -9.   -8.5  -8.   -7.5
  -7.   -6.5  -6.   -5.5  -5.   -4.5  -4.   -3.5  -3.   -2.5  -2.   -1.5
  -1.   -0.5   0.    0.5   1.    1.5   2.    2.5   3.    3.5   4.    4.5
   5.    5.5   6.    6.5   7.    7.5   8.    8.5   9.    9.5  10.   10.5
  11.   11.5  12.   12.5  13.   13.5  14.   14.5  15.   15.5  16.   16.5
  17.   17.5  18.   18.5  19.   19.5  20.   20.5  21.   21.5  22.   22.5
  23.   23.5  24.   24.5  25. ]
```

The advantage in using `linspace` is that we can have non-integer steps between each array element, something that cannot be accomplished directly via the `range` command.

7.9 maximum

Given two arrays of equal shape, finds their element-wise maximum values.

Here is an example:

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.maximum(a, b))
```

Giving us our output:

```
[[5 6]
 [7 8]]
```

7.10 meshgrid

Allows the user to quickly create two mesh arrays from two linearly-spaced 1-D arrays. As an example, let's create two arrays using `linspace` and pass them to meshgrid and see what is created:

```
import numpy as np

a = np.linspace(0, 5, 6)
b = np.linspace(0, 3, 4)
X, Y = np.meshgrid(a, b)
```

Printing `X` give us the following result:

```
[[0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]
 [0. 1. 2. 3. 4. 5.]]
```

While `Y` give us a similar array to `X`, only in terms of the y-direction:

```
[[0. 0. 0. 0. 0. 0.]
 [1. 1. 1. 1. 1. 1.]
 [2. 2. 2. 2. 2. 2.]
 [3. 3. 3. 3. 3. 3.]]
```

Meshgrids come in handy when dealing with functions of multiple variables, and works for any number of dimensions, just be sure to have enough variables ready for the resulting output[16].

[16] For instance, we would have needed an `X`, `Y`, and `Z` variable if we'd chosen to pass three linspaces as meshgrid arguments.

7.11 minimum

Given two arrays of equal shape, finds their element-wise minimum values.

Here is an example:

```
import numpy as np

a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6], [7, 8]])
print(np.minimum(a, b))
```

Giving us our output:

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

7.12 ones

Creates an array of ones in the desired shape (in the form of an integer or array of integers).

To create a 2×2 array of ones, we can use the script below:

```
import numpy as np

a = np.ones((2, 2))
```

Printing our array leaves us with our desired output:

```
[[1. 1.]
 [1. 1.]]
```

7.13 ones_like

Creates an array of ones in the shape of an input array.

To create a 2×2 array of ones with `ones_like`, we can use the script below:

```
import numpy as np

a = np.ones_like([[1, 2], [3, 4]])
```

Printing our array leaves us with this output:

```
[[1. 1.]
 [1. 1.]]
```

7.14 prod

Takes the product of an array for a given `axis` – if the `axis` is not defined, it will take the total product over all its elements.

If we have a simple 1-D array, using `prod` is straightforward; this process gets complex quickly as we increase our array dimension, so it is recommended that the user be familiar with Section 4.5.

For a 3-D array, we can take the product over its axis `0` with the following:

```
1 import numpy as np
2
3 a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
4 print(np.prod(a, axis = 0))
```

Since its zero-th axis refers to the set of sub-arrays highest up in the hierarchy, we end up with the products of `[[1, 2], [3, 4]]` and `[[5, 6], [7, 8]]`, giving us the following output:

```
[[ 5 12]
 [21 32]]
```

7.15 sum

Takes the sum of an array for a given `axis` – if the `axis` is not defined, it will take the total sum over all its elements.

If we have a simple 1-D array, using `sum` is straightforward; this process gets complex quickly as we increase our array dimension, so it is recommended that the user be familiar with Section 4.5.

For a 3-D array, we can take the sum over its axis `0` with the following:

```
1 import numpy as np
2
3 a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
4 print(np.sum(a, axis = 0))
```

Since its zero-th axis refers to the set of sub-arrays highest up in the hierarchy, we end up with the sums of `[[1, 2], [3, 4]]` and `[[5, 6], [7, 8]]`, giving us the following output:

```
[[ 6  8]
 [10 12]]
```

7.16 zeros

Creates an array of zeros in the desired shape (in the form of an integer or array of integers).

To create a 2×2 array of zeros, we can use the script below:

```
1 import numpy as np
```

```
2
3 a = np.zeros((2, 2))
```

Printing our array leaves us with our desired output:

```
[[0. 0.]
 [0. 0.]]
```

7.17 zeros_like

Creates an array of zeros in the shape of an input array.

To create a 2×2 array of zeros with `zeros_like`, we can use the script below:

```
1 import numpy as np
2
3 a = np.zeros_like([[1, 2], [3, 4]])
```

Printing our array leaves us with this output:

```
[[0. 0.]
 [0. 0.]]
```