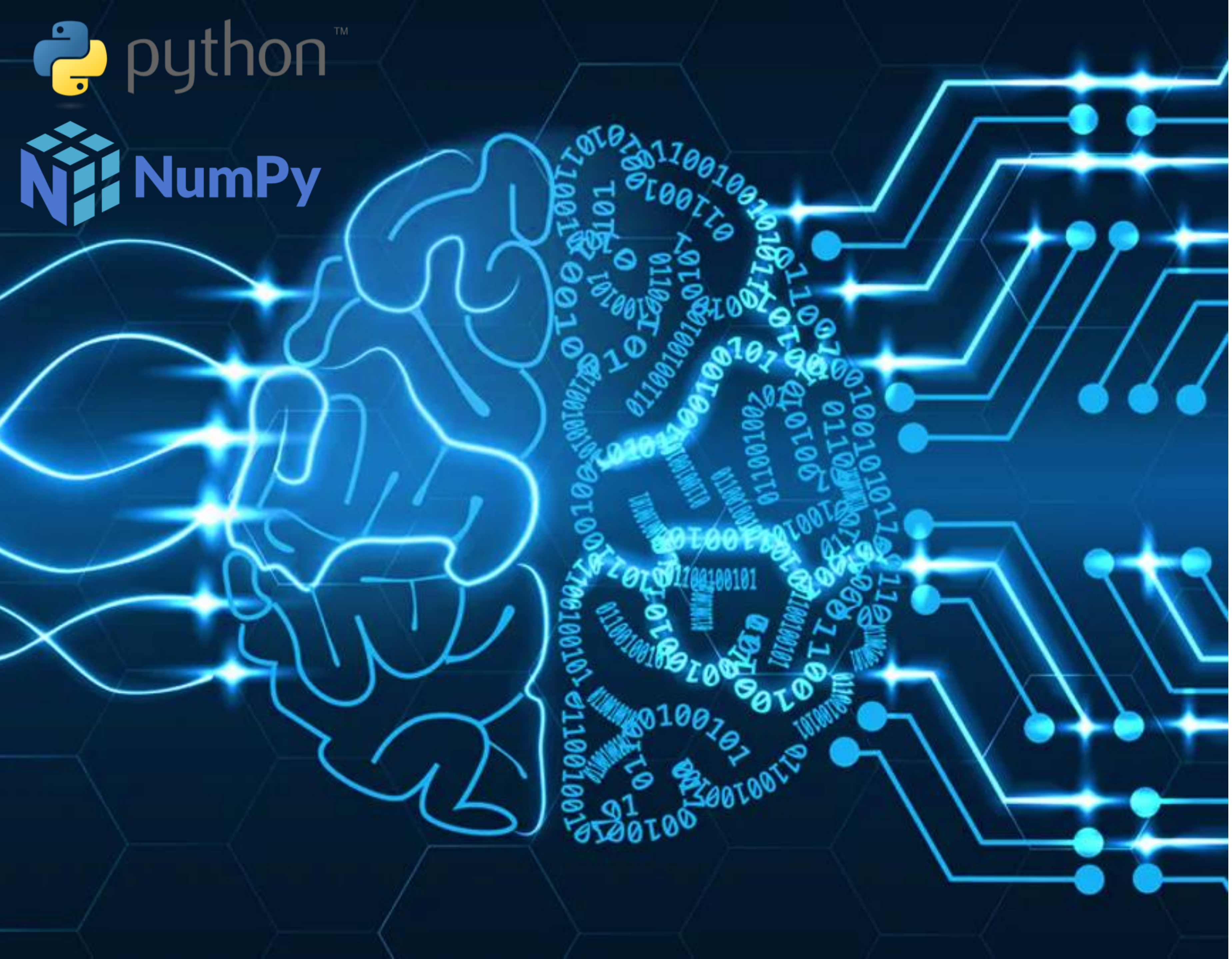


یادگیری ماشین
آشنایی با کتابخانه Numpy

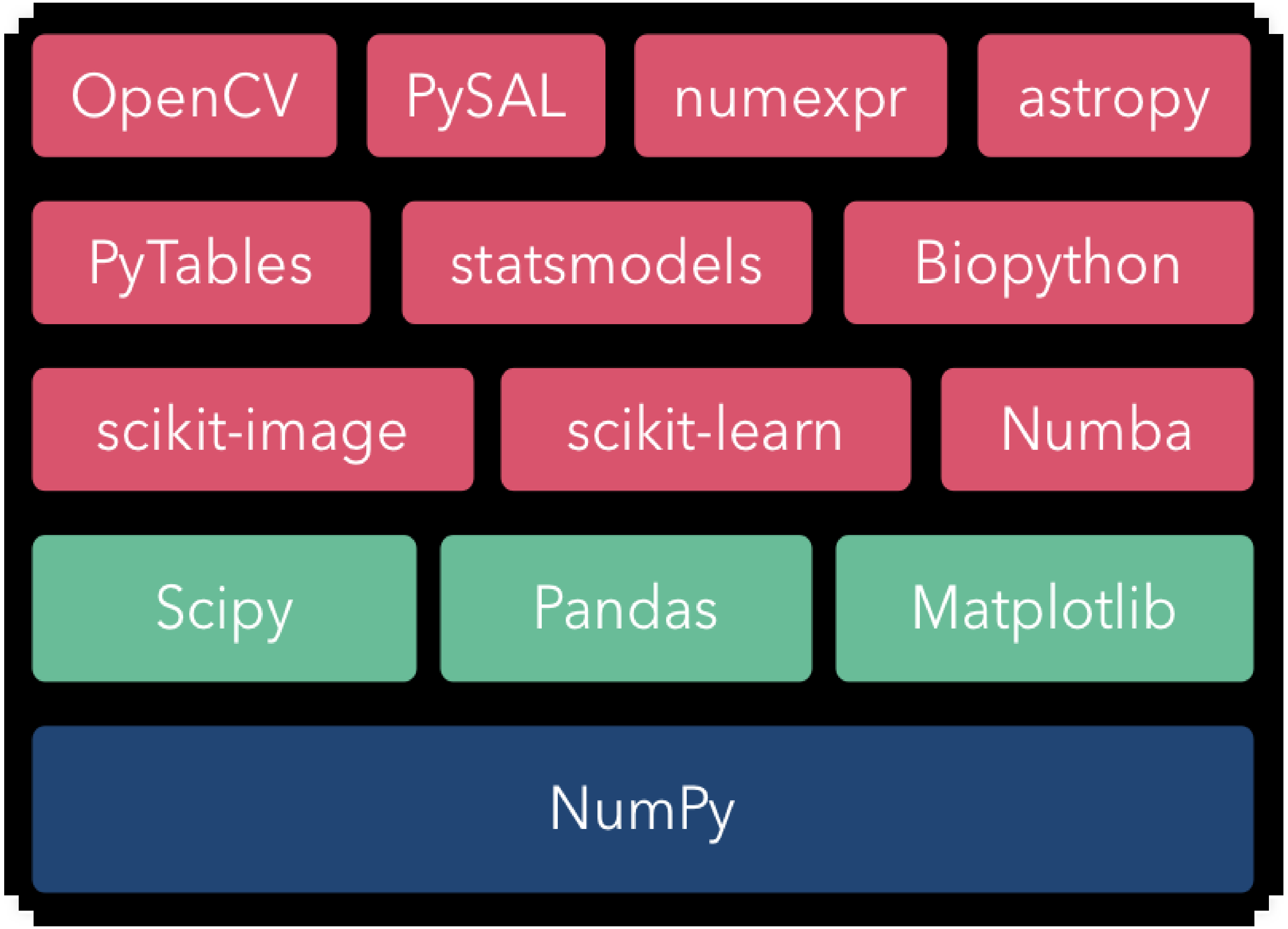
دکتر مهدی شریف‌زاده

رضا الوندی – امیرحسین محمودی

بهمن ۱۴۰۱

بسم الله الرحمن الرحیم

NumPy is the foundation of the python scientific stack



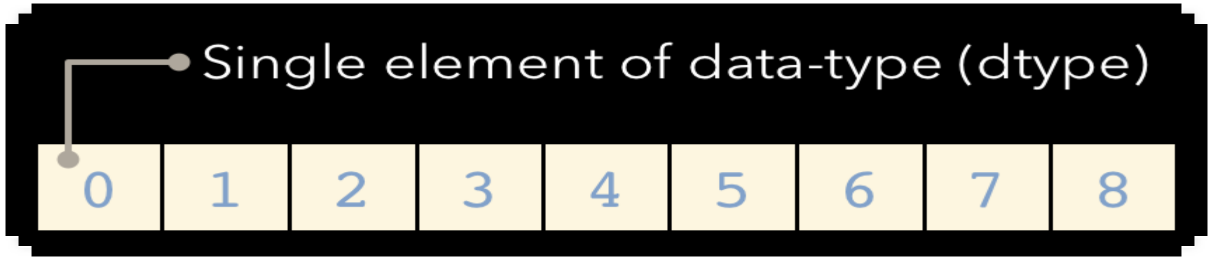
NumPy

Numpy is the backbone of Machine Learning in Python. It is one of the most important libraries in Python for numerical computations. It adds support to core Python for multi-dimensional arrays (and matrices) and fast vectorized operations on these arrays.

The present day NumPy library is a successor of an early library, Numeric, which was created by Jim Hugunin and some other developers. Travis Oliphant, Anaconda’s president and co-founder, took the Numeric library as a base and added a lot of modifications, to launch the present day NumPy library in 2005. It is a major open source project and is one of the most popular Python libraries. It’s used in almost all Machine Learning and scientific computing libraries.

Numpy ndarray

- NumPy is a Python C extension library for array-oriented computing
  - Efficient
  - In-memory
  - Contiguous (or Strided)
  - Homogeneous (but types can be algebraic)



- NumPy is suited to many applications
  - Image processing
  - Signal processing
  - Linear algebra
  - A plethora of others

Numpy ndarray

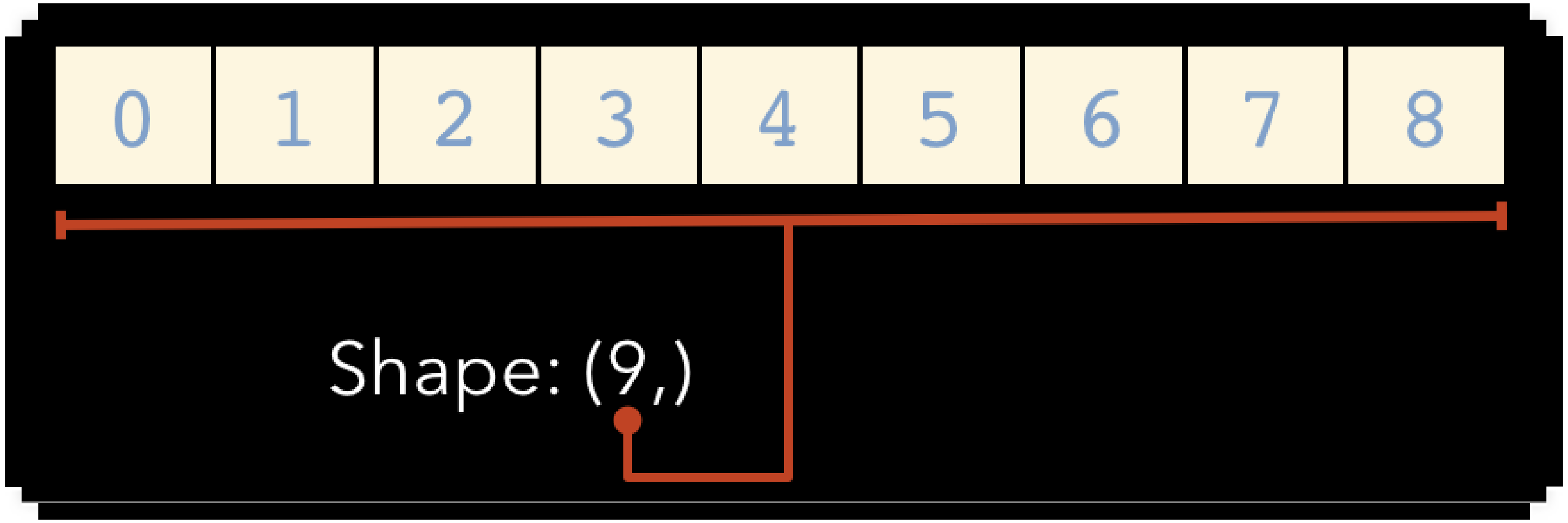
All of the numeric functionality of numpy is orchestrated by two important constituents of the numpy package, ndarray and Ufuncs (Universal function). Numpy ndarray is a multi-dimensional array object which is the core data container for all of the numpy operations. Universal functions are the functions which operate on ndarrays in an element by element fashion.

Numpy ndarray

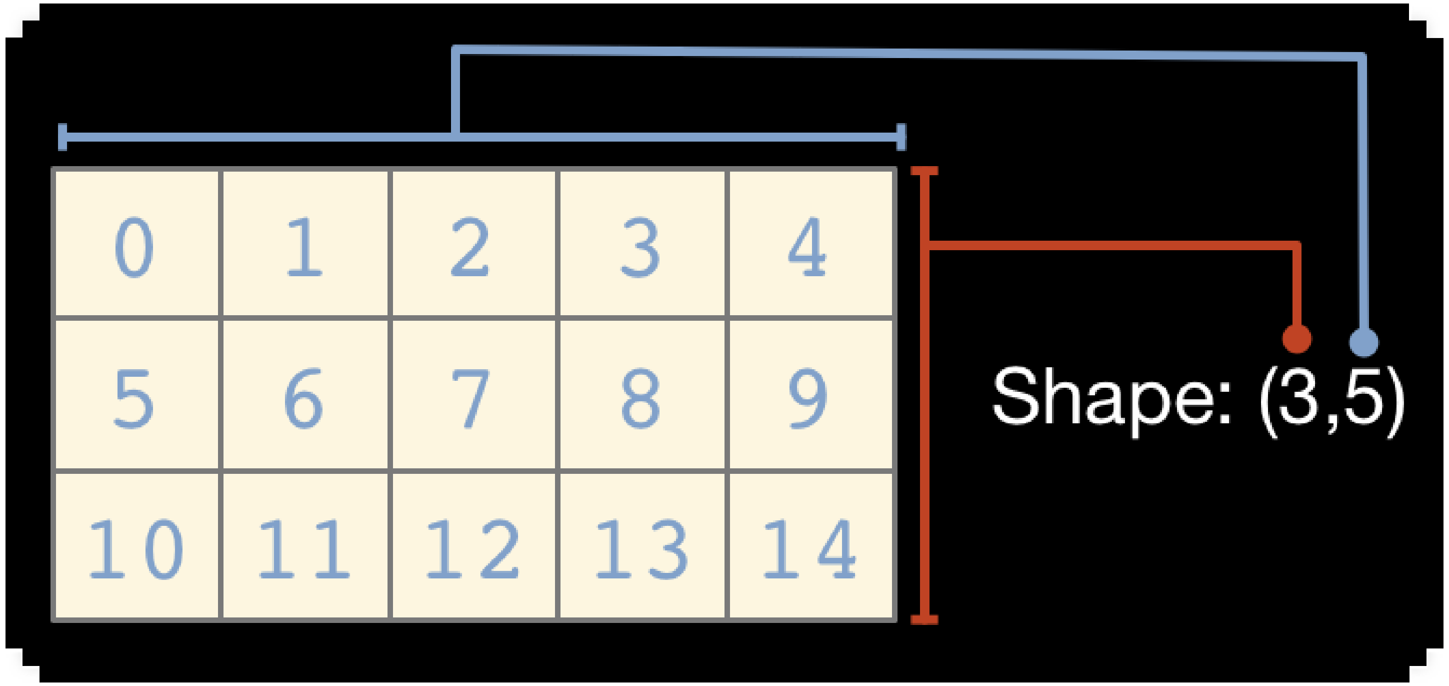
These are the lesser known members of the numpy package and we will try to give a brief introduction to them in the later stage of this section. We will mostly be learning about ndarrays in subsequent sections. (We will refer to them as arrays from now on for simplicity’s sake.) Arrays (or matrices) are one of the fundamental representations of data. Mostly an array will be of a single data type (homogeneous) and possibly multi-dimensional sometimes. The numpy ndarray is a generalization of the same.

Array Shape

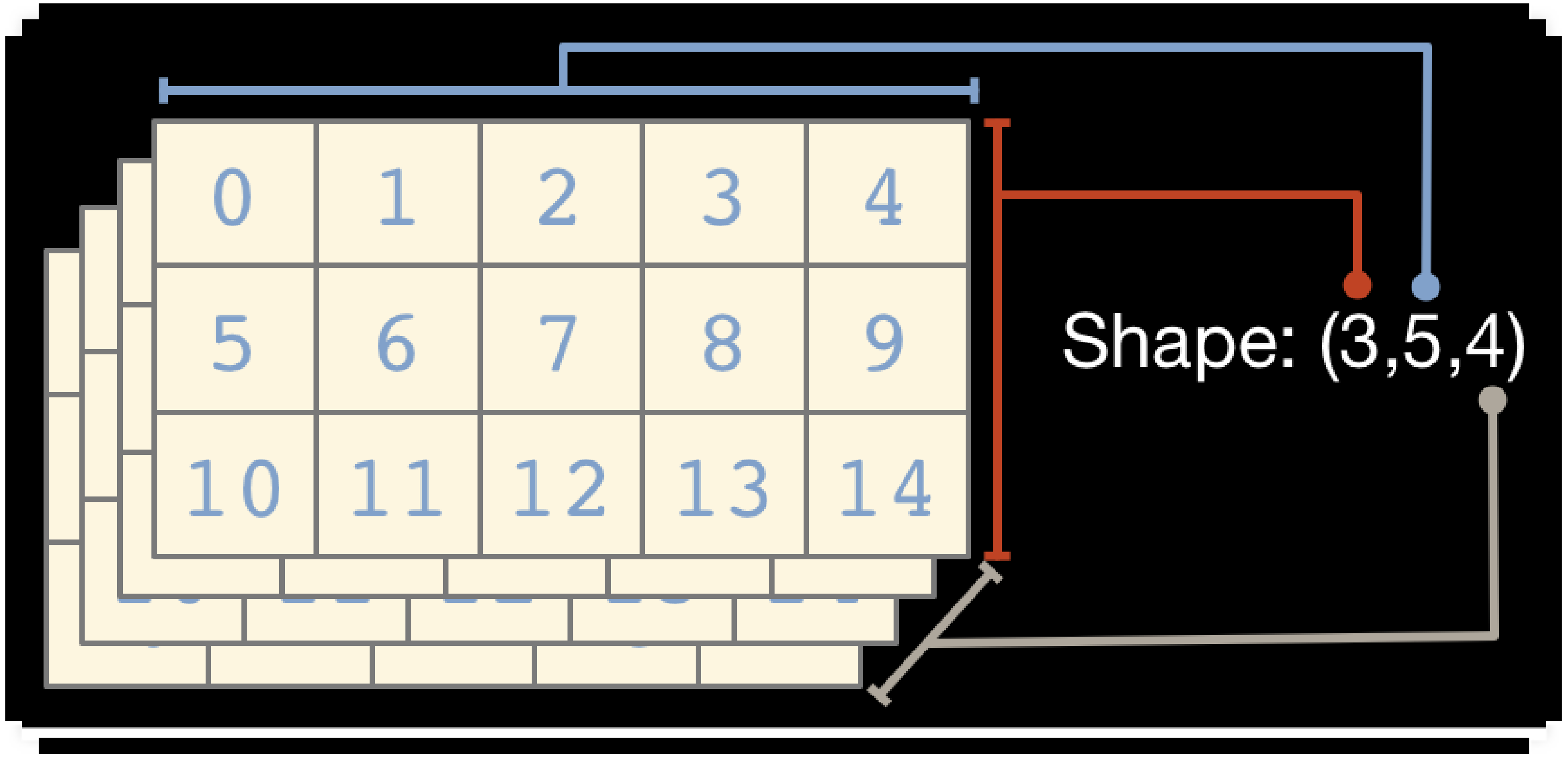
One dimensional arrays have a 1-tuple for their shape



Two dimensional arrays have a 2-tuple...

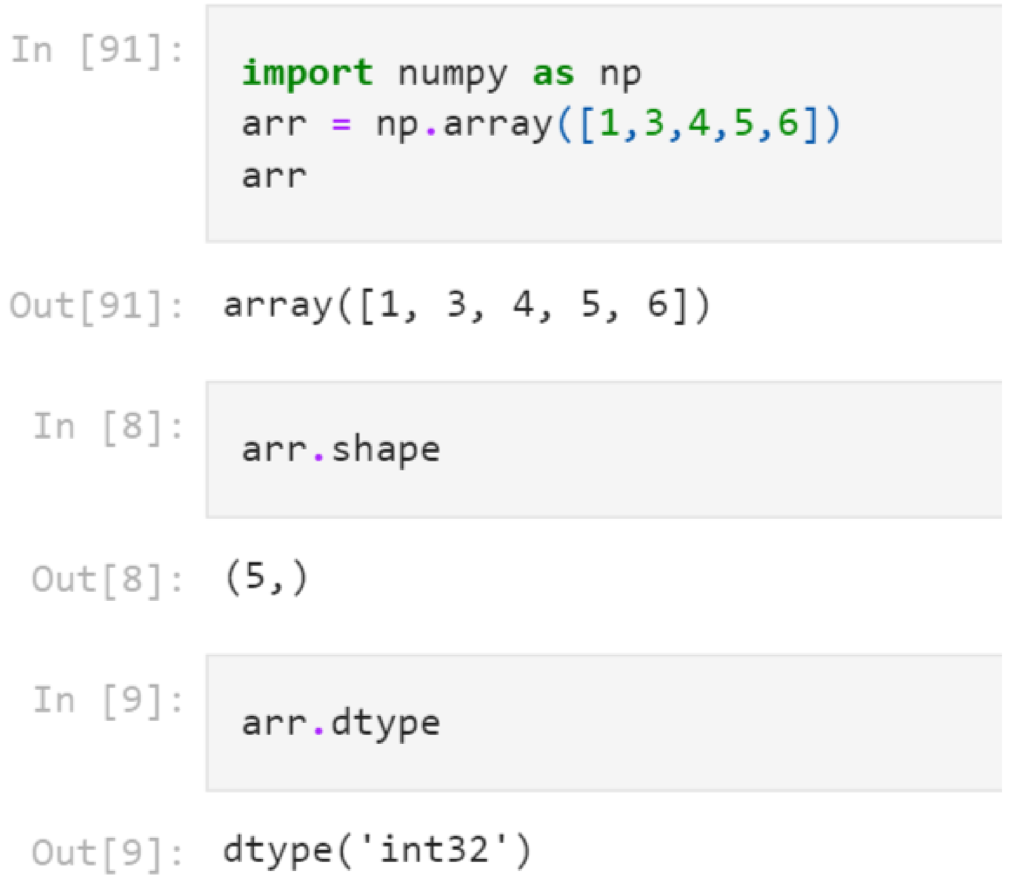


And so on...



Numpy ndarray

Let’s get started with the introduction by creating an array.



```
In [91]:  import numpy as np
           arr = np.array([1,3,4,5,6])
           arr

Out[91]:  array([1, 3, 4, 5, 6])

In [8]:   arr.shape

Out[8]:   (5,)

In [9]:   arr.dtype

Out[9]:   dtype('int32')
```

Array Element Type (dtype)

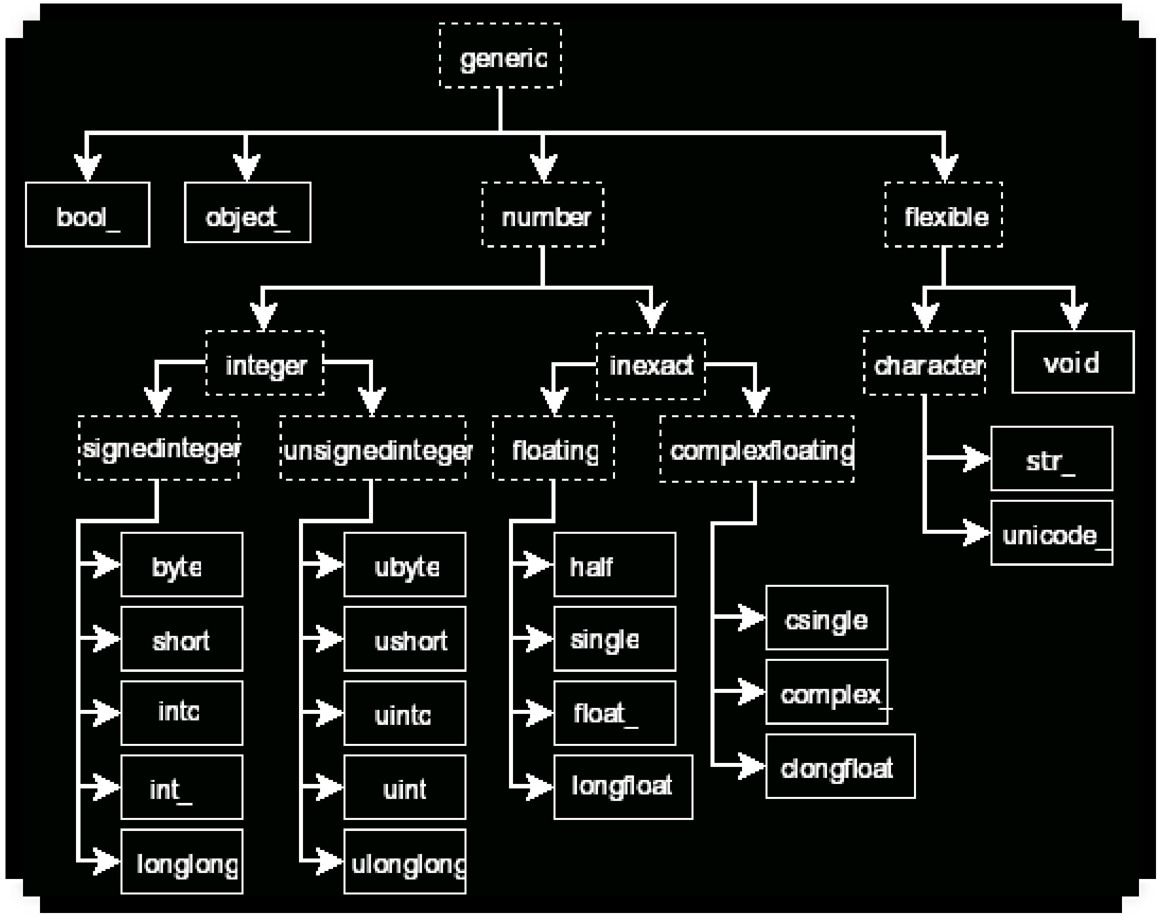
- NumPy arrays comprise elements of a single data type The type object is accessible through the .dtype attribute

Here are a few of the most important attributes of dtype objects

- dtype.byteorder — big or little endian
- dtype.itemsize — element size of this dtype
- dtype.name — a name for this dtype object
- dtype.type — type object used to create scalars

There are many others...

NumPy Builtin dtype Hierarchy



np.datetime64is a new addition in NumPy 1.7

Numpy ndarray

In the previous example, we created a one-dimensional array from a normal list containing integers. The shape attribute of the array object will tell us about the dimensions of the array. The data type was picked up from the elements as they were all integers the data type is int32. One important thing to keep in mind is that all the elements in an array must have the same data type. If you try to initialize an array in which the elements are mixed, i.e. you mix some strings with the numbers then all of the elements will get converted into a string type and we won't be able to perform most of the numpy operations on that array. So a simple rule of thumb is dealing only with numeric data.

```
arr = np.array([1,'st','er',3])
arr.dtype
```

```
dtype('<U11')
```

Creating Arrays

Arrays can be created in multiple ways in numpy. One of the ways was demonstrated earlier to create a single-dimensional array. Similarly we can stack up multiple lists to create a multidimensional array

```
arr = np.array([[1,2,3],[2,4,6],[8,8,8]])
arr.shape
```

```
(3, 3)
```

```
arr
```

```
array([[1, 2, 3],
       [2, 4, 6],
       [8, 8, 8]])
```

Creating Arrays

In addition to this we can create arrays using a bunch of special functions provided by numpy.

np.zeros: Creates a matrix of specified dimensions containing only zeroes:

```
arr = np.zeros((2,4))
arr
```

```
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

np.ones: Creates a matrix of specified dimension containing only ones:

```
arr = np.ones((2,4))
arr
```

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

np.identity: Creates an identity matrix of specified dimensions:

```
arr = np.identity(3)
arr
```

```
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

Creating Arrays (genfromtext)

One of the functions that we can use to read data from text file to a numpy array is genfromtext. This function can open a text file and read in data delimited by any character. (delimiter for a comma separated file is “,”). Since it is not our preferred way of retrieving data, we will give a brief example of the function here.

```
from io import BytesIO
b = BytesIO(b"2,23,33\n32,42,63.4\n35,77,12")
arr = np.genfromtxt(b, delimiter=",")
arr
```

```
array([[  2. ,   23. ,   33. ],
       [ 32. ,   42. ,   63.4],
       [ 35. ,   77. ,   12. ]])
```

Accessing Array Elements

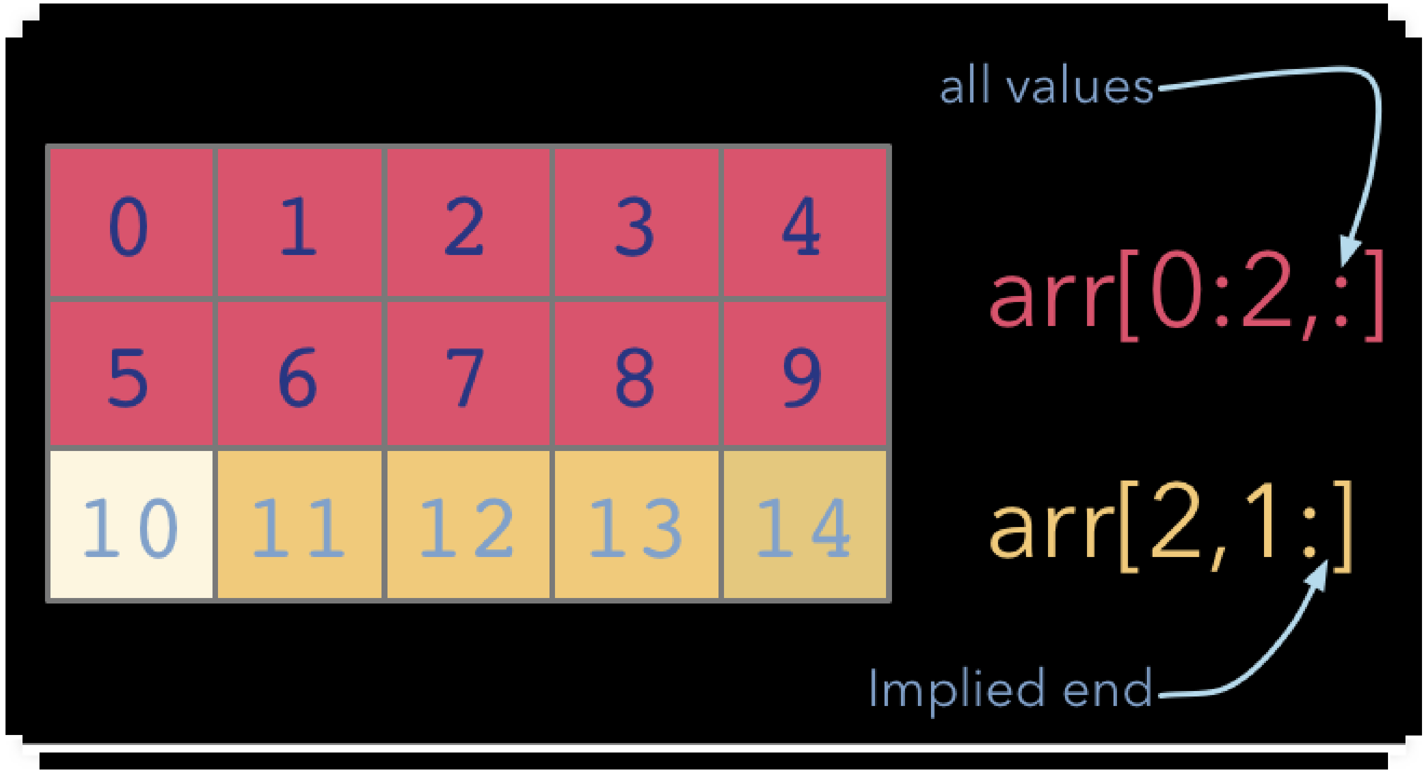
Once we have created an array by reading in our data, the next important part is to access that data using a wide variety of mechanisms. Numpy provides a lot of ways in which array elements can be accessed. We will try to give the most popular useful ways that facilitate this.

Basic Indexing and Slicing

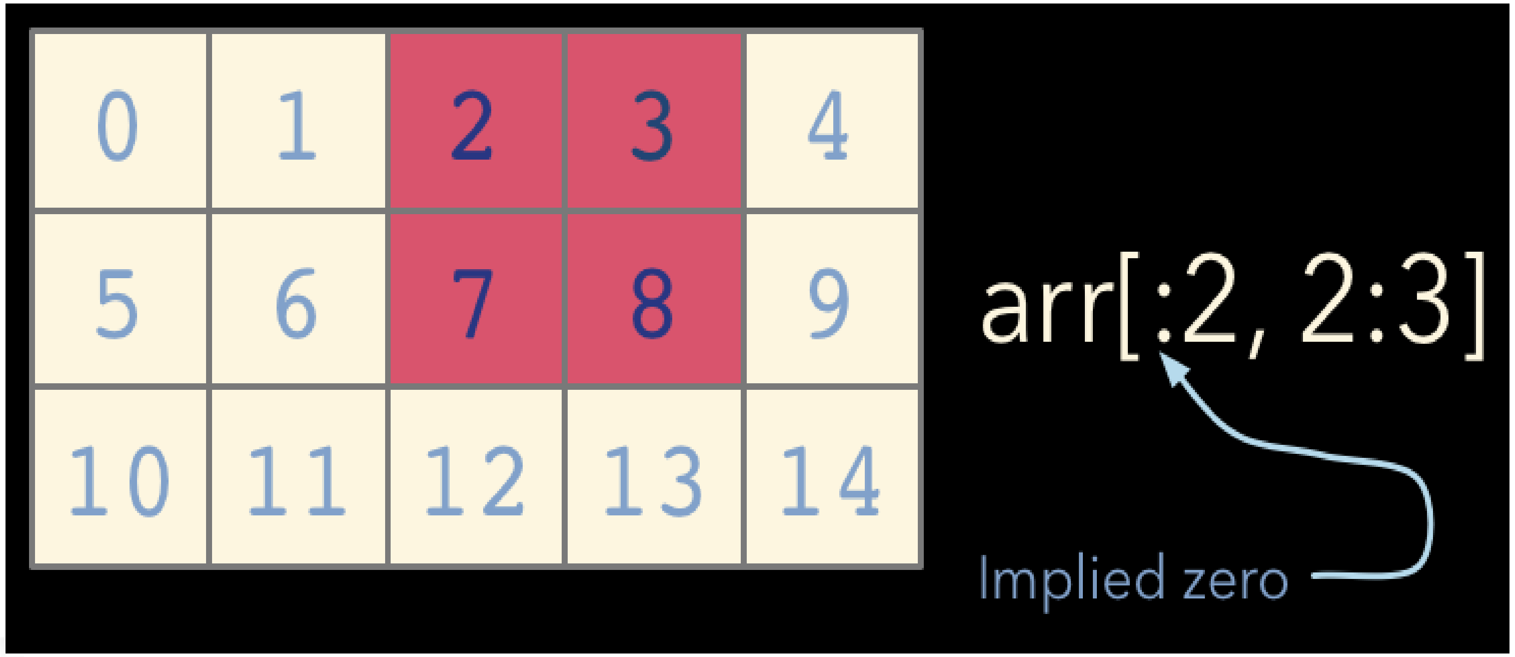
Ndarray can leverage the basic indexing operations that are followed by the list class, i.e. list object [obj].
If the obj is not an ndarray object, then the indexing is said to be basic indexing.

one important point to remember is that basic indexing will always return a view of the original array. it means that it will only refer to the original array and any change in values will be reflected in the original array also.

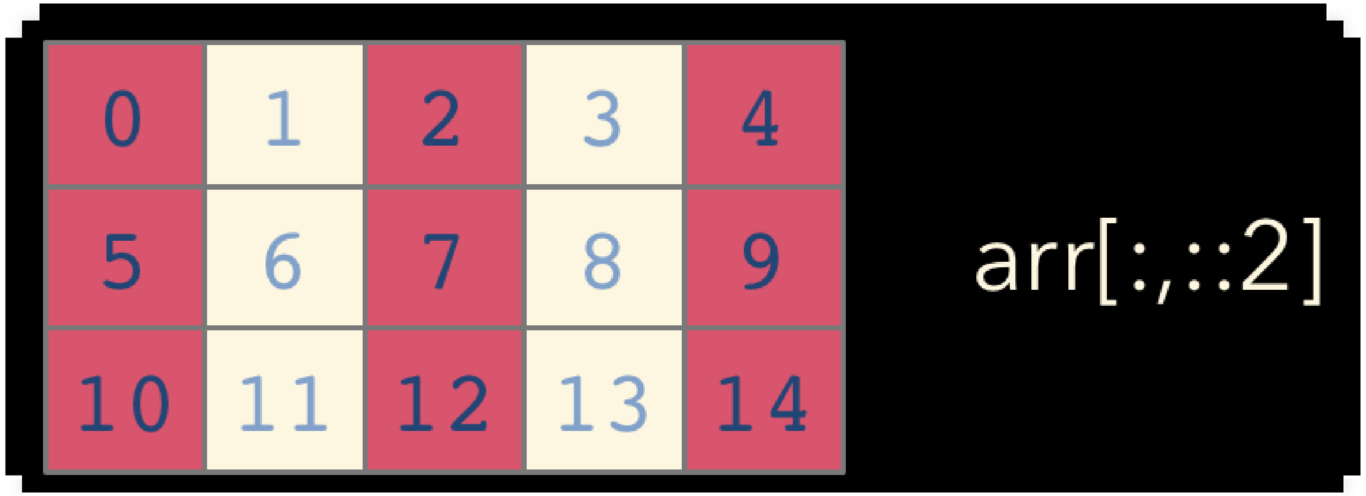
Indexing and Slicing

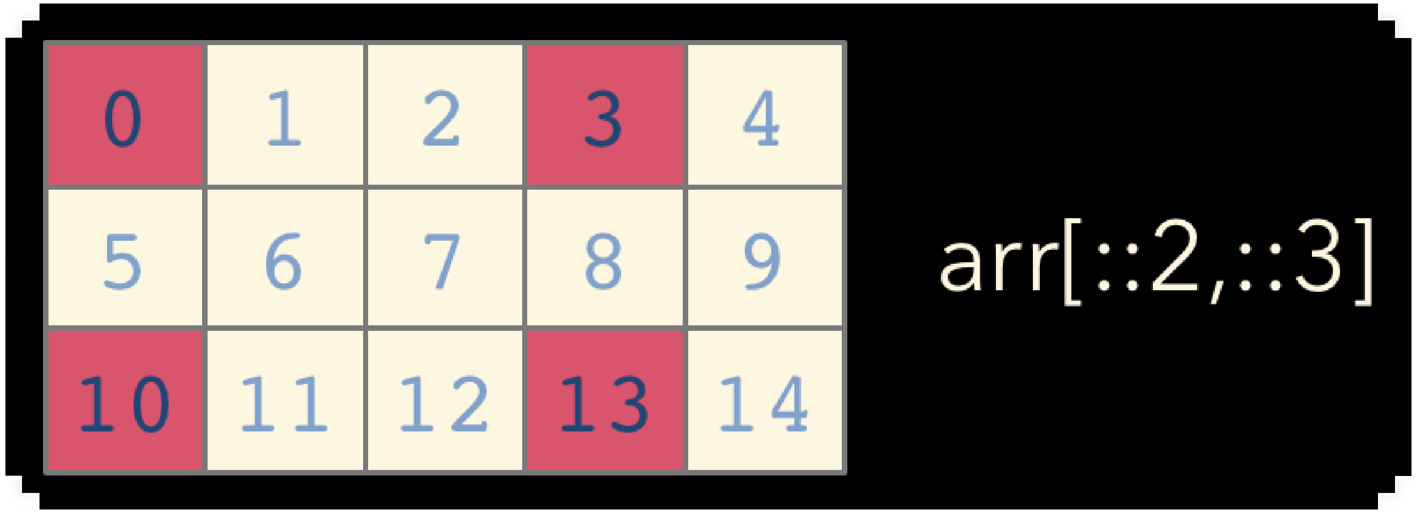


Indexing and Slicing



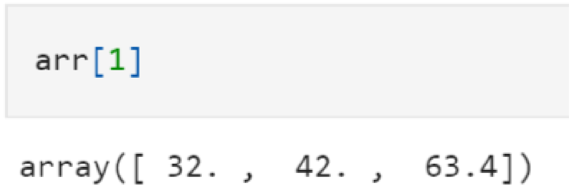
NumPy array indices can also take an optional stride





Accessing Array Elements

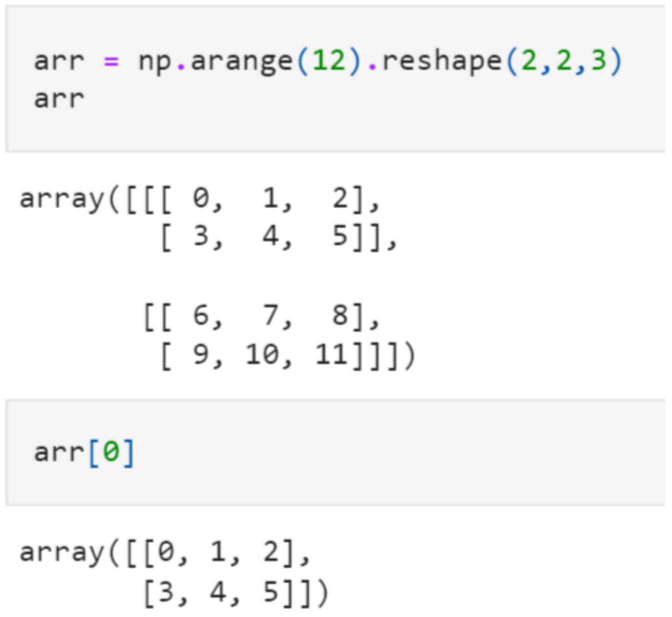
For example, if we want to access the complete second row of the array in one of the earlier examples,
we can simply refer to it using arr[1].



```
arr[1]
```

```
array([ 32. ,  42. ,  63.4])
```

This access becomes interesting in the case of an array having more than two dimensions. Consider the following code snippet.



```
arr = np.arange(12).reshape(2,2,3)
arr
```

```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

```
arr[0]
```

```
array([[0, 1, 2],
       [3, 4, 5]])
```

Here we see that using a similar indexing scheme as above, we get an array having one lesser dimension than the original array

Accessing Array Elements

The next important concept in accessing arrays is the concept of slicing arrays. Suppose we want to have a collection of elements only instead of all the elements. Then we can use slicing to access the elements. We will demonstrate the concept with a one-dimensional array.

```
arr = np.arange(10)
arr[5:]
```

```
array([5, 6, 7, 8, 9])
```

```
arr[5:8]
```

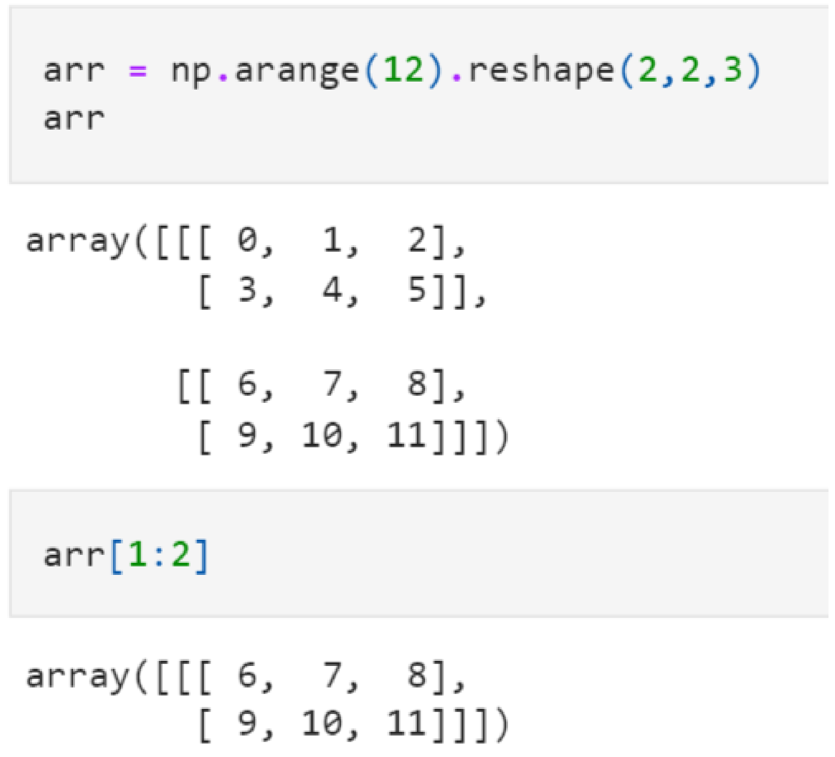
```
array([5, 6, 7])
```

```
arr[:-5]
```

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

Accessing Array Elements

If the number of dimensions in the object supplied is less than the dimension of the array being accessed then the colon (:) is assumed for all the dimensions. Consider the following example



```
arr = np.arange(12).reshape(2,2,3)
arr
```

```
array([[[ 0,  1,  2],
        [ 3,  4,  5]],

       [[ 6,  7,  8],
        [ 9, 10, 11]]])
```

```
arr[1:2]
```

```
array([[[ 6,  7,  8],
        [ 9, 10, 11]]])
```

Accessing Array Elements

Another way to access an array is to use dots (...) based indexing. Suppose in a three-dimensional array we want to access the value of only one column. We can do it in two ways.

```
arr = np.arange(27).reshape(3,3,3)
arr
```

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

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

Accessing Array Elements

Now if we want to access the third column, we can use two different notations to access that column:

```
arr[:,:,2]
```

```
array([[ 2,  5,  8],
       [11, 14, 17],
       [20, 23, 26]])
```

We can also use a dot notation in the following way. Both of the methods gets us the same value but the dot notation is concise. The dot notation stands for as many colons as required to complete an indexing operation.

```
arr[...,2]
```

```
array([[ 2,  5,  8],
       [11, 14, 17],
       [20, 23, 26]])
```

Advanced Indexing

The difference in advanced indexing and basic indexing comes from the type of object being used to reference the array. If the object is an ndarray object (data type int or bool) or a non-tuple sequence object or a tuple object containing an ndarray (data type integer or bool), then the indexing being done on the array is said to be advanced indexing.

Advanced indexing will always return the copy of the original array data.

Advanced Indexing

Integer array indexing:

This advanced indexing occurs when the reference object is also an array. The simplest type of indexing is when we provide an array that's equal in dimensions to the array being accessed. For example:

```
arr = np.arange(9).reshape(3,3)
arr
```

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

In this example we have provided an array in which the first part identifies the rows we want to access and the second identifies the columns which we want to address. This is quite similar to providing a collective element-wise address

```
arr[[0,1,2],[1,0,0]]
```

```
array([1, 3, 6])
```

Advanced Indexing

Boolean indexing:

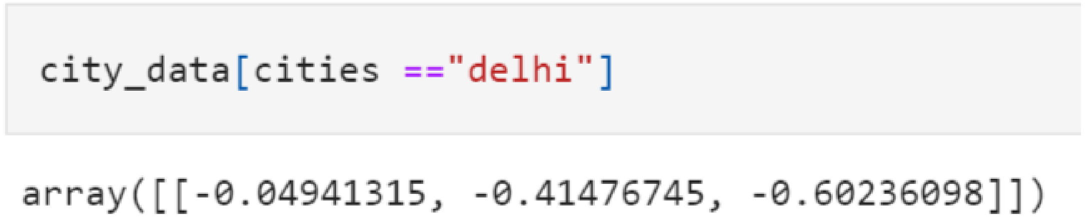
This advanced indexing occurs when the reference object is an array of Boolean values. This is used when we want to access data based on some conditions, in that case, Boolean indexing can be used. We will illustrate it with an example. Suppose in one array, we have the names of some cities and in another array, we have some data related to those cities

```
cities = np.array(["delhi","banglaore","mumbai","chennai","bhopal"])
city_data = np.random.randn(5,3)
city_data
```

```
array([[-0.04941315, -0.41476745, -0.60236098],
       [-1.75033842,  0.62559942, -0.58148095],
       [ 0.43502897, -0.06588454, -0.40865494],
       [-0.53978394, -0.7317352 , -0.66959325],
       [ 0.45550659, -0.53018559, -0.2241479 ]])
```

Advanced Indexing

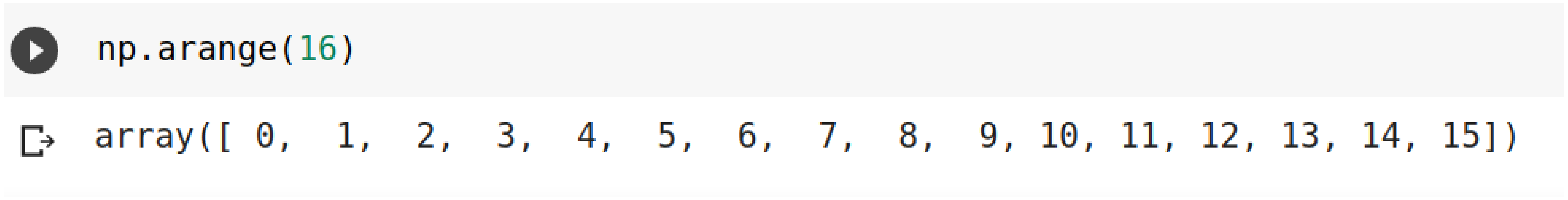
We can also use Boolean indexing for selecting some elements of an array that satisfy a particular condition. For example, in the previous array suppose we want to only select non-zero elements. We can do that easily using the following code.



```
city_data[cities =="delhi"]
```

```
array([[-0.04941315, -0.41476745, -0.60236098]])
```

We can generate a sequence of integer numbers using np.arange function.

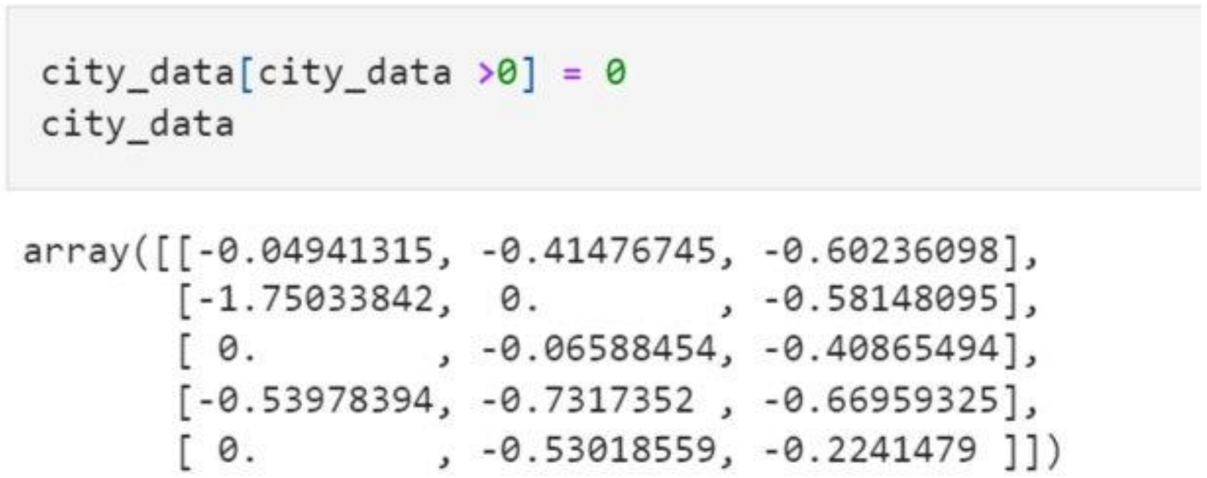


```
np.arange(16)
```

```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

Advanced Indexing

We observe that the shape of the array is not maintained so we directly cannot always use this indexing method. But this method is quite useful in doing conditional data substitution. Suppose in the previous case, we want to substitute all the non-zero values with 0. We can achieve that operation by the following code.



```
city_data[city_data >0] = 0
city_data
```

```
array([[-0.04941315, -0.41476745, -0.60236098],
       [-1.75033842,  0.        , -0.58148095],
       [ 0.        , -0.06588454, -0.40865494],
       [-0.53978394, -0.7317352 , -0.66959325],
       [ 0.        , -0.53018559, -0.2241479 ]])
```

Operations on Arrays

At the start of this section, we mentioned the concept of Universal functions (Ufuncs). In this sub-section, we learn some of the functionalities provided by those functions. Most of the operations on the numpy arrays is achieved by using these functions. Numpy provides a rich set of functions that we can leverage for various operations on arrays. We cover some of those functions in brief, but we recommend you to always refer to the official documentation of the project to learn more and leverage them in your own projects.

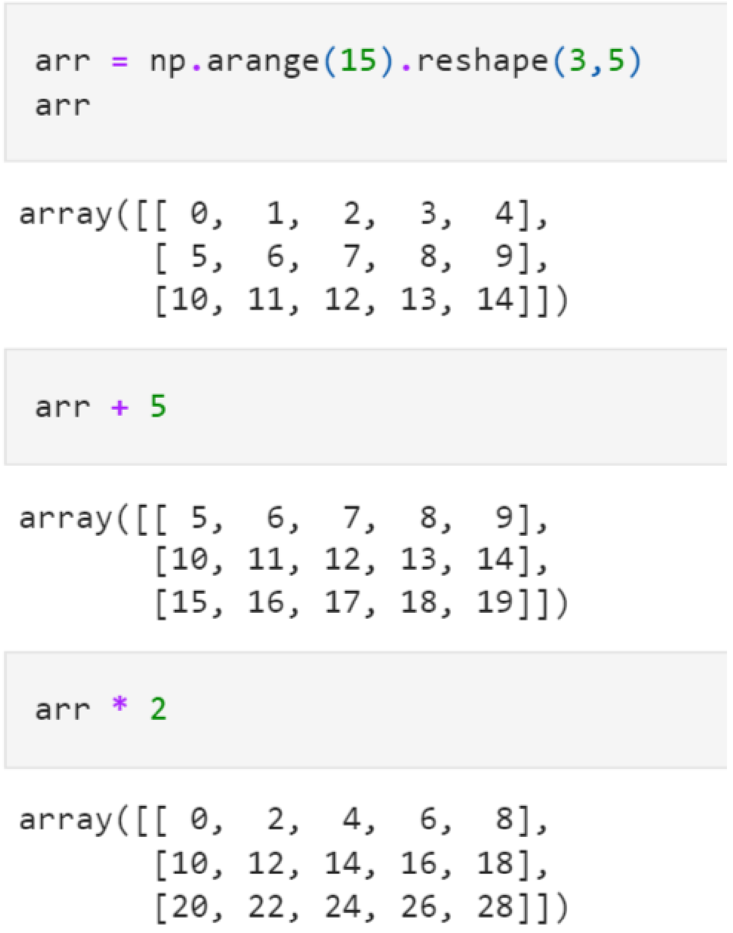
Operations on Arrays

Universal functions are functions that operate on arrays in an element by element fashion. The implementation of Ufunc is vectorized, which means that the execution of Ufuncs on arrays is quite fast. The Ufuncs implemented in the numpy package are implemented in compiled C code for speed and efficiency. But it is possible to write custom functions by extending the numpy.ufunc class of the numpy package.

Operations on Arrays

Ufuncs are simple and easy to understand once you are able to relate the output they produce on a particular array.

We see that the standard operators when used in conjunction with arrays work element-wise.



```
arr = np.arange(15).reshape(3,5)
arr
```

```
array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])
```

```
arr + 5
```

```
array([[ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14],
       [15, 16, 17, 18, 19]])
```

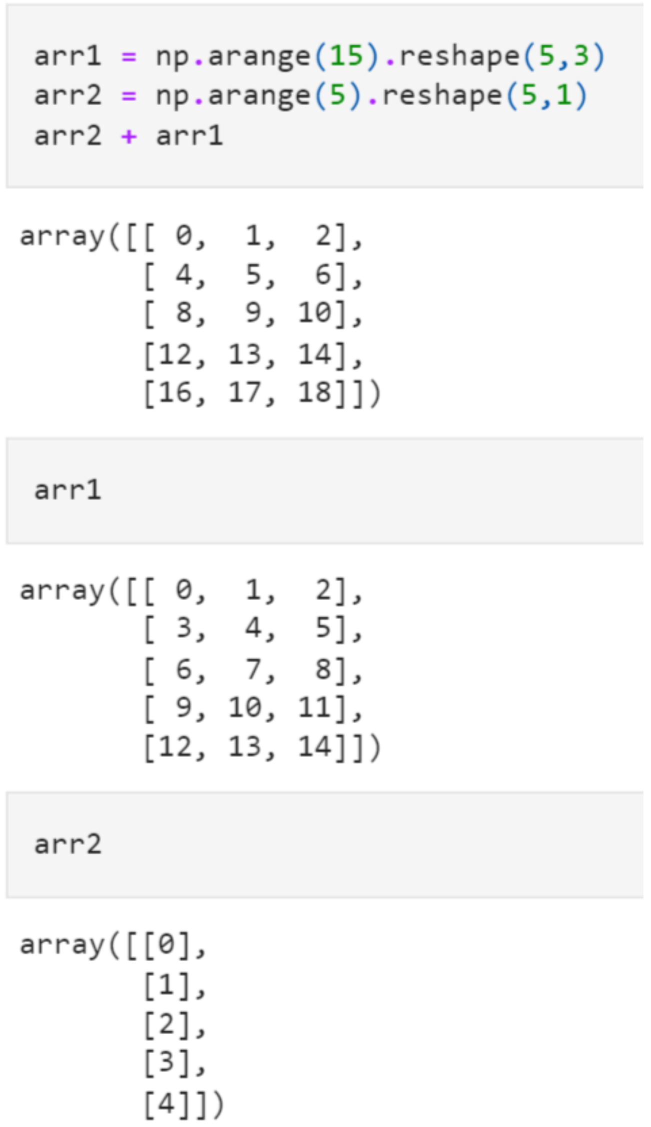
```
arr * 2
```

```
array([[ 0,  2,  4,  6,  8],
       [10, 12, 14, 16, 18],
       [20, 22, 24, 26, 28]])
```

Operations on Arrays

Some Ufuncs will take two arrays as input and output a single array, while a rare few will output two arrays also.

Here we see that we were able to add up two arrays even when they were of different sizes. This is achieved by the concept of *broadcasting*.



```
arr1 = np.arange(15).reshape(5,3)
arr2 = np.arange(5).reshape(5,1)
arr2 + arr1
```

```
array([[ 0,  1,  2],
       [ 4,  5,  6],
       [ 8,  9, 10],
       [12, 13, 14],
       [16, 17, 18]])
```

```
arr1
```

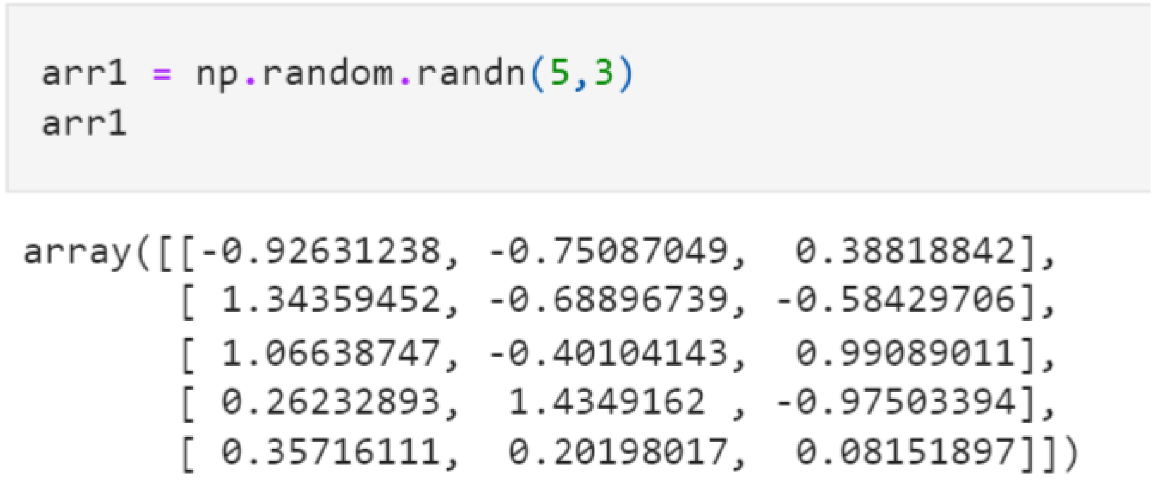
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11],
       [12, 13, 14]])
```

```
arr2
```

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

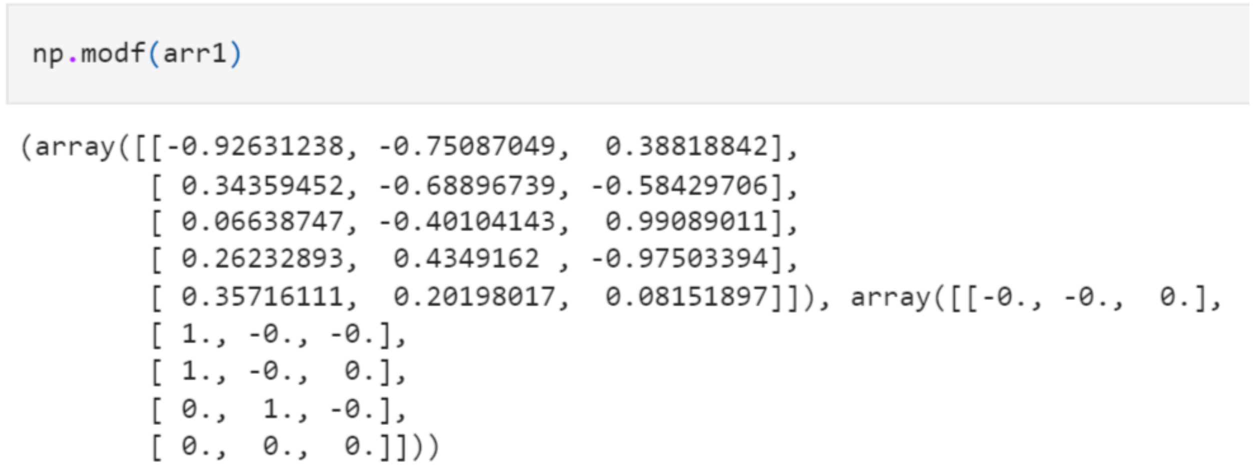
Operations on Arrays

We will conclude this brief discussion on operations on arrays by demonstrating a function that will return two arrays.



```
arr1 = np.random.randn(5,3)
arr1
```

```
array([[-0.92631238, -0.75087049,  0.38818842],
       [ 1.34359452, -0.68896739, -0.58429706],
       [ 1.06638747, -0.40104143,  0.99089011],
       [ 0.26232893,  1.4349162 , -0.97503394],
       [ 0.35716111,  0.20198017,  0.08151897]])
```



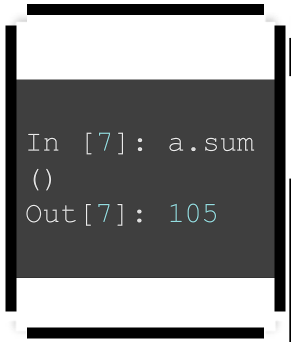
```
np.modf(arr1)
```

```
(array([[-0.92631238, -0.75087049,  0.38818842],
        [ 0.34359452, -0.68896739, -0.58429706],
        [ 0.06638747, -0.40104143,  0.99089011],
        [ 0.26232893,  0.4349162 , -0.97503394],
        [ 0.35716111,  0.20198017,  0.08151897]]), array([[-0., -0.,  0.],
        [ 1., -0., -0.],
        [ 1., -0.,  0.],
        [ 0.,  1., -0.],
        [ 0.,  0.,  0.]]))
```

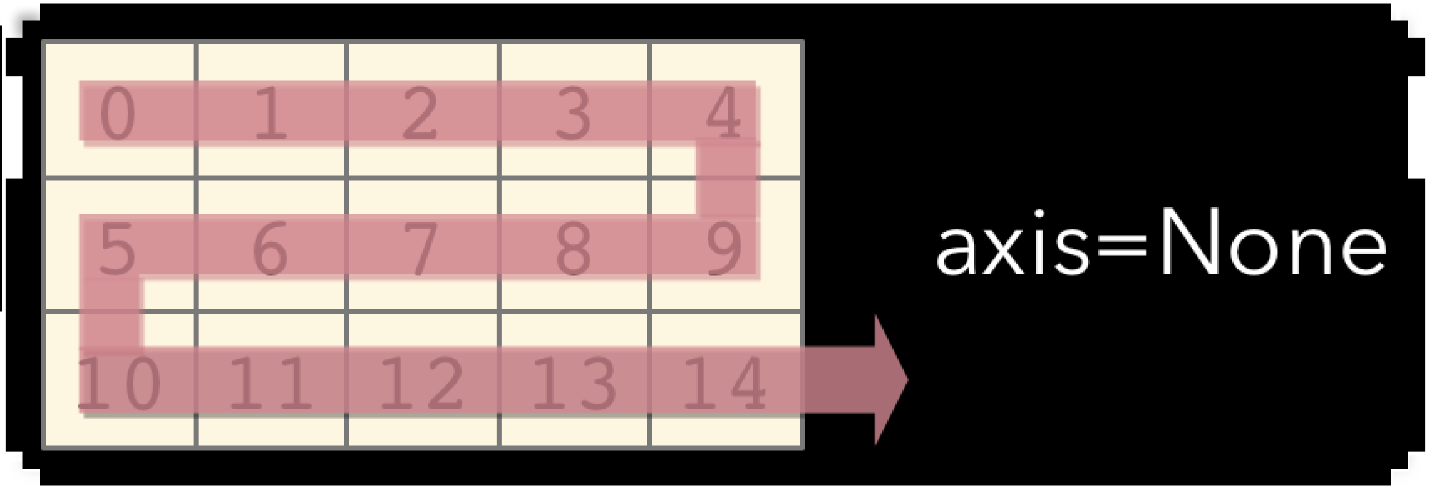
The function modf will return the fractional and the integer part of the input supplied to it. Hence it will return two arrays of the same size.

Axis

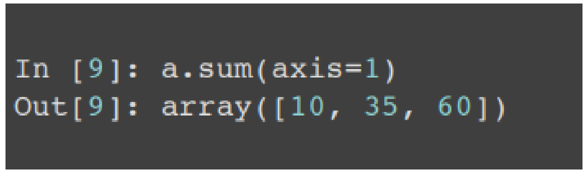
Array method reductions take an optional axisparameter that specifies over which axes to reduce axis=Nonereduces into a single scalar



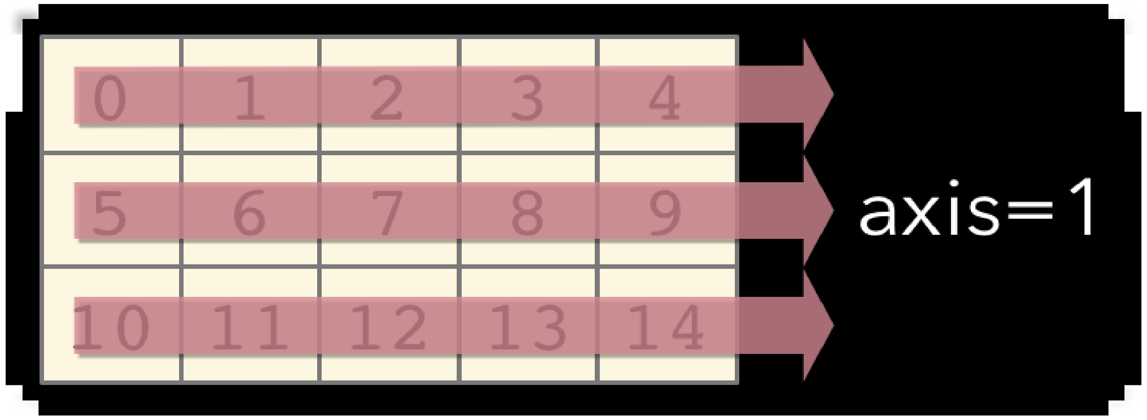
```
In [7]: a.sum
()
Out[7]: 105
```



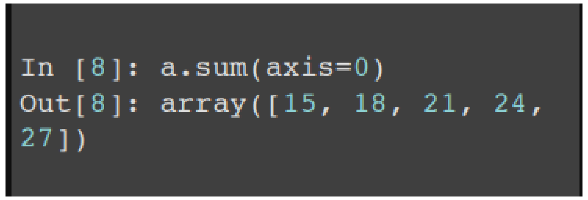
`axis=1` reduces into the zeroth dimension



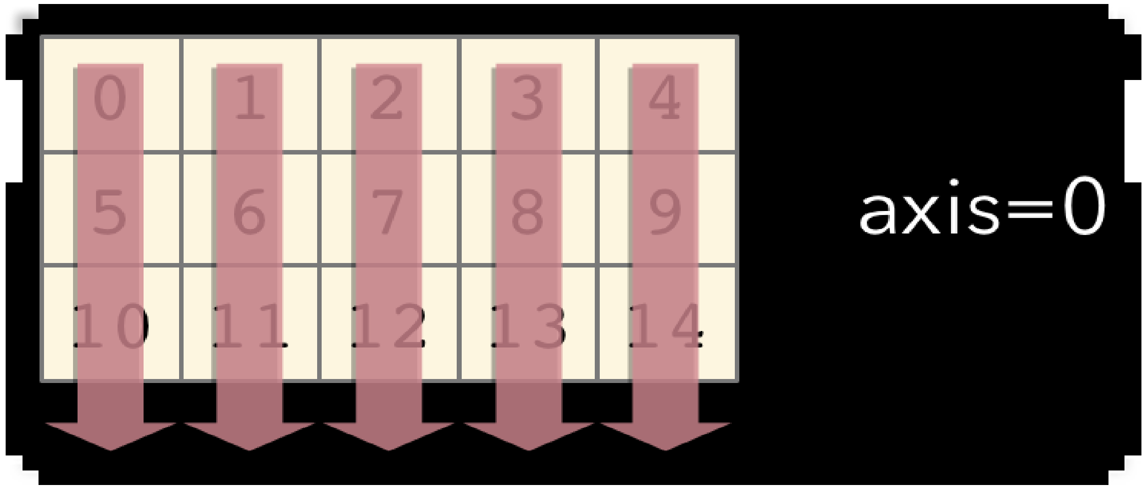
```
In [9]: a.sum(axis=1)
Out[9]: array([10, 35, 60])
```



`axis=0` reduces into the first dimension

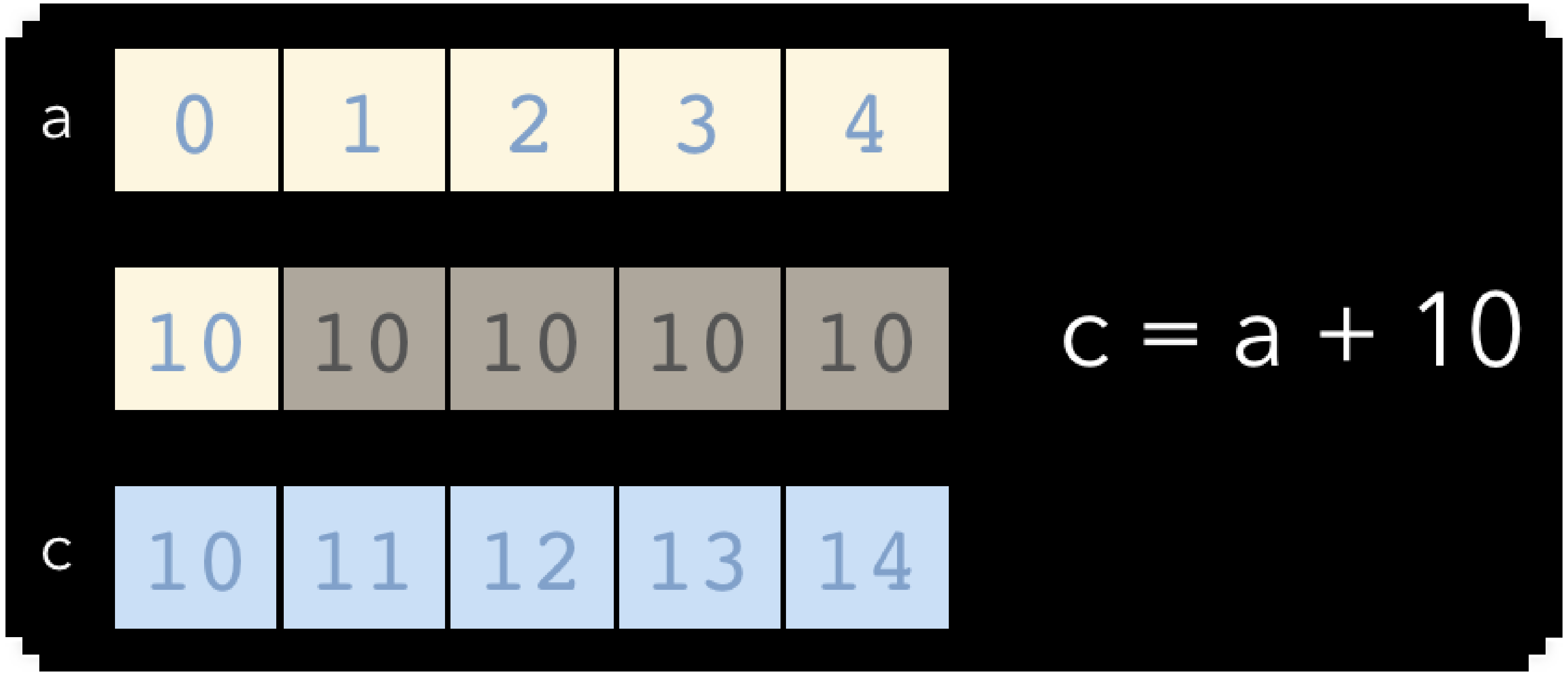


```
In [8]: a.sum(axis=0)
Out[8]: array([15, 18, 21, 24,
27])
```



Broadcasting

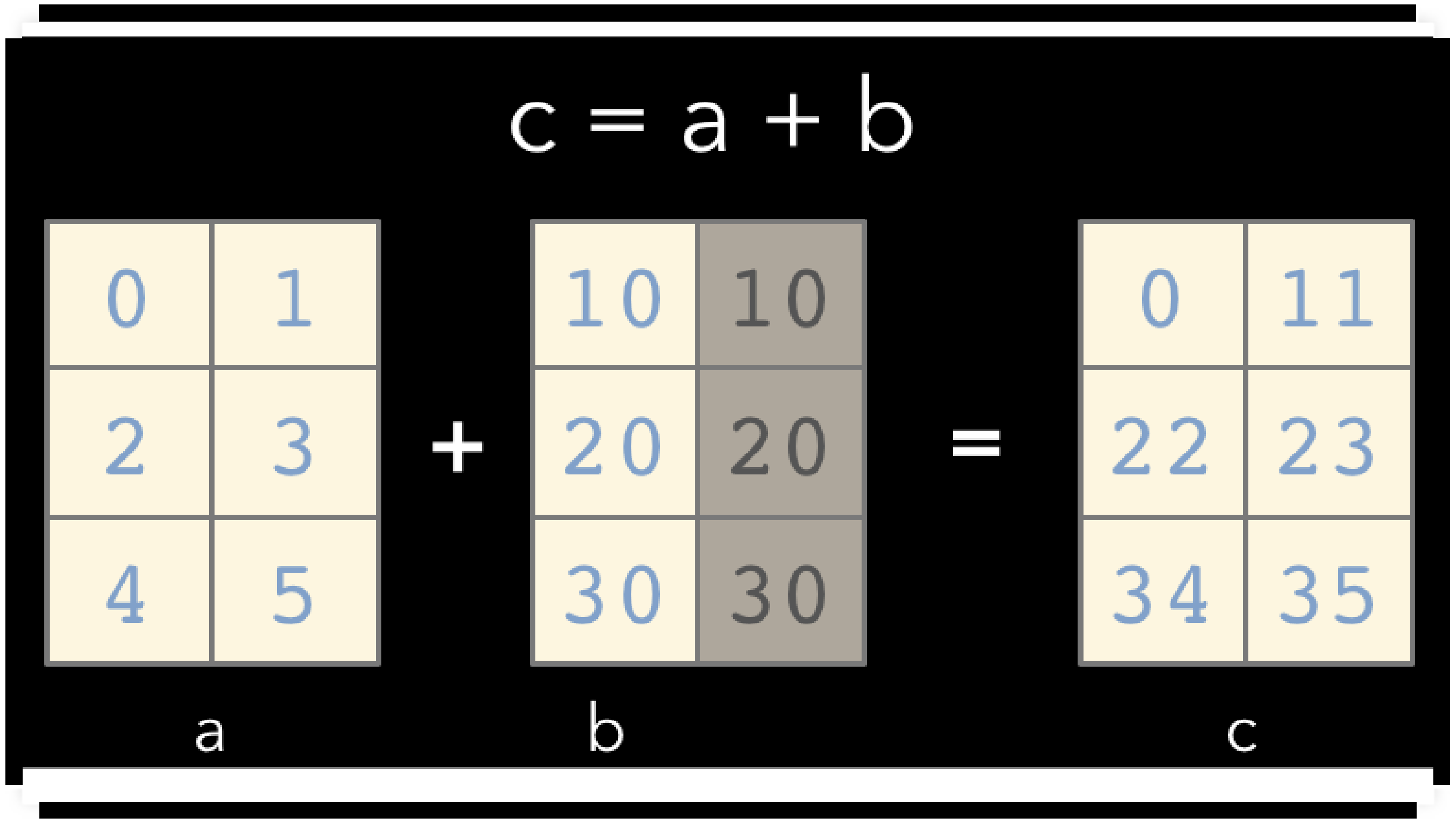
A key feature of NumPy is broadcasting, where arrays with different, but compatible shapes can be used as arguments to ufuncs



In this case an array scalar is broadcast to an array with shape (5,)

Broadcasting

A slightly more involved broadcasting example in two dimensions



Here an array of shape (3, 1)is broadcast to an array with shape (3, 2)

Broadcasting Rules

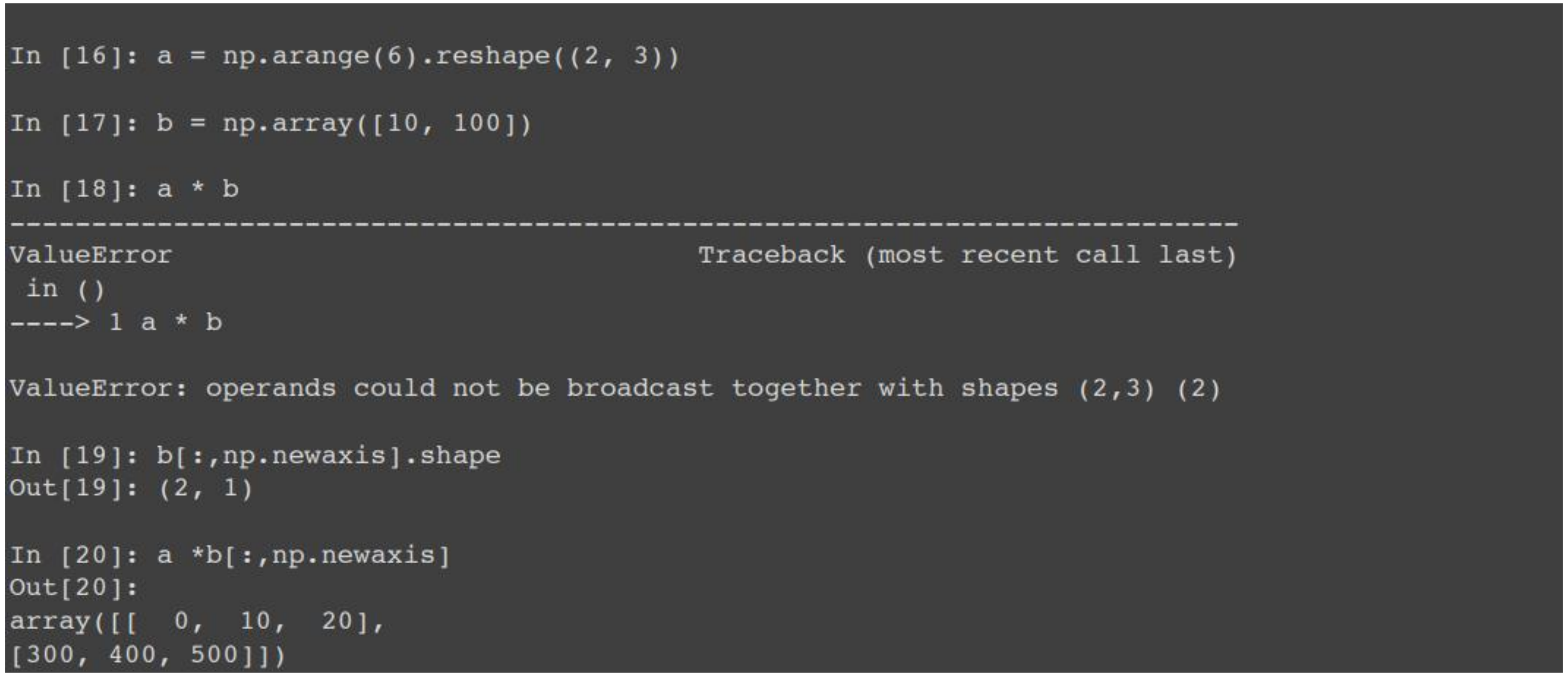
In order for an operation to broadcast, the size of all the trailing dimensions for both arrays must either:

be equal OR be one

A	(1d	array):							3
B	(2d	array):					2	x	3
Result	(2d	array):					2	x	3
A	(2d	array):					6	x	1
B	(3d	array):			1	x	6	x	4
Result	(3d	array):			1	x	6	x	4
A	(4d	array):	3	x	1	x	6	x	1
B	(3d	array):			2	x	1	x	4
Result	(4d	array):	3	x	2	x	6	x	4

Square Peg in a Round Hole

If the dimensions do not match up, np.newaxismay be useful



```
In [16]: a = np.arange(6).reshape((2, 3))

In [17]: b = np.array([10, 100])

In [18]: a * b
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
 in ()
----> 1 a * b

ValueError: operands could not be broadcast together with shapes (2,3) (2)

In [19]: b[:,np.newaxis].shape
Out[19]: (2, 1)

In [20]: a *b[:,np.newaxis]
Out[20]:
array([[  0,  10,  20],
[300, 400, 500]])
```

Linear Algebra Using numpy

Linear algebra is an integral part of the domain of Machine Learning. Most of the algorithms we will deal with can be concisely expressed using the operations of linear algebra. Numpy was initially built to provide the functions similar to MATLAB and hence linear algebra functions on arrays were always an important part of it. In this section, we learn a bit about performing linear algebra on ndarrays using the functions implemented in the numpy package.

Linear Algebra Using numpy

One of the most widely used operations in linear algebra is the dot product. This can be performed on two compatible (brush up on your matrices and array skills if you need to know which arrays are compatible for a dot product) ndarrays by using the dot function.

Similarly, there are functions implemented for finding different products of matrices like inner, outer, and so on. Another popular matrix operation is transpose of a matrix. This can be easily achieved by using the T function.

```
A = np.array([[1,2,3],[4,5,6],[7,8,9]])
B = np.array([[9,8,7],[6,5,4],[1,2,3]])
A.dot(B)
```

```
array([[ 24,  24,  24],
       [ 72,  69,  66],
       [120, 114, 108]])
```

```
A = np.arange(15).reshape(3,5)
A.T
```

```
array([[ 0,  5, 10],
       [ 1,  6, 11],
       [ 2,  7, 12],
       [ 3,  8, 13],
       [ 4,  9, 14]])
```

Linear Algebra Using numpy

Oftentimes, we need to find out decomposition of a matrix into its constituents factors. This is called matrix factorization. This can be achieved by the appropriate functions. A popular matrix factorization method is SVD factorization (covered briefly in Chapter 1 concepts), which returns decomposition of a matrix into three different matrices. This can be done using linalg.svd function.

```
np.linalg.svd(A)
```

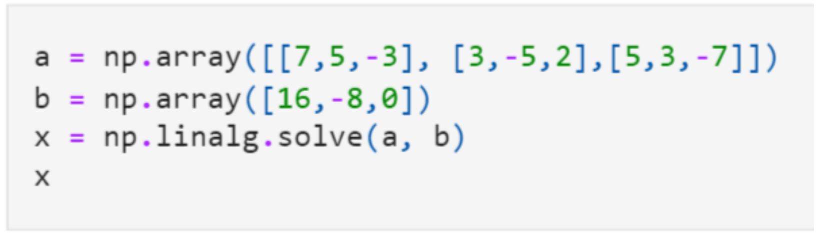
```
(array([[-0.15425367,  0.89974393,  0.40824829],
        [-0.50248417,  0.28432901, -0.81649658],
        [-0.85071468, -0.3310859 ,  0.40824829]]),
 array([  3.17420265e+01,   2.72832424e+00,   4.58204637e-16]),
 array([[-0.34716018, -0.39465093, -0.44214167, -0.48963242, -0.53712316],
        [-0.69244481, -0.37980343, -0.06716206,  0.24547932,  0.55812069],
        [ 0.33717486, -0.77044776,  0.28661392,  0.38941603, -0.24275704],
        [-0.36583339,  0.32092943, -0.08854543,  0.67763613, -0.54418674],
        [-0.39048565,  0.05843412,  0.8426222 , -0.29860414, -0.21196653]]))
```

Linear Algebra Using numpy

Linear algebra is often also used to solve a system of equations. Using the matrix notation of system of equations and the provided function of numpy, we can easily solve such a system of equation. Consider the system of equations:

$$7x + 5y - 3z = 16$$
$$3x - 5y + 2z = -8$$
$$5x + 3y - 7z = 0$$

This can be represented as two matrices: the coefficient matrix (a in the example) and the constants
vector (b in the example).

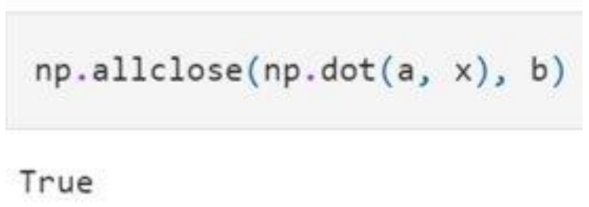


```
a = np.array([[7,5,-3], [3,-5,2],[5,3,-7]])
b = np.array([16,-8,0])
x = np.linalg.solve(a, b)
x
```

```
array([ 1.,  3.,  2.])
```

Linear Algebra Using numpy

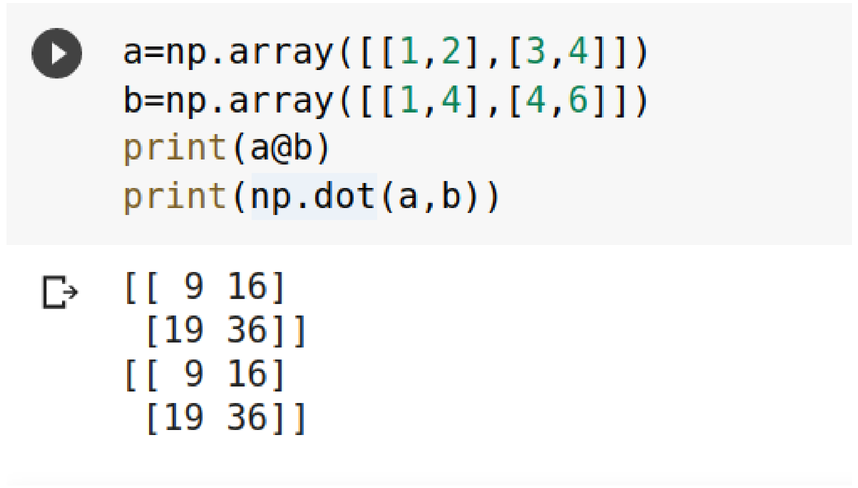
We can also check if the solution is correct using the np.allclose function



```
np.allclose(np.dot(a, x), b)
```

```
True
```

We can do matrix multiplication using np.dot function or using '@' which is fancier

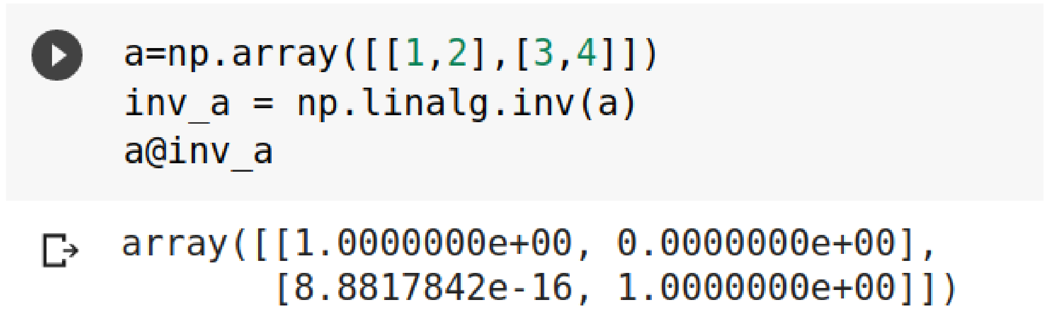


```
a=np.array([[1,2],[3,4]])
b=np.array([[1,4],[4,6]])
print(a@b)
print(np.dot(a,b))
```

```
[[ 9 16]
 [19 36]]
[[ 9 16]
 [19 36]]
```

Linear Algebra Using numpy

We can find matrix inverse using np.linalg.inv function



```
a=np.array([[1,2],[3,4]])
inv_a = np.linalg.inv(a)
a@inv_a
```

```
array([[1.0000000e+00, 0.0000000e+00],
       [8.8817842e-16, 1.0000000e+00]])
```

Challenge on Numpy

1. Create an array of random numbers between 10 and 100 in shape of (50,50). (hint: use np.random.randint)
2. Reshape the array to (5,5,10,10)
3. Change elements greater than 25 to 300
4. Reshape the array to (50,50)
5. Get elements in address [24,12] and [23,45] in a list with only one instruction

References

- http://docs.scipy.org/doc/numpy/reference/
- http://docs.scipy.org/doc/numpy/user/index.html
- http://www.scipy.org/Tentative_NumPy_Tutorial
- http://www.scipy.org/Numpy_Example_List
- Practical Machine Learning with Python(A Problem-Solver's Guide to Building Real-World Intelligent Systems)