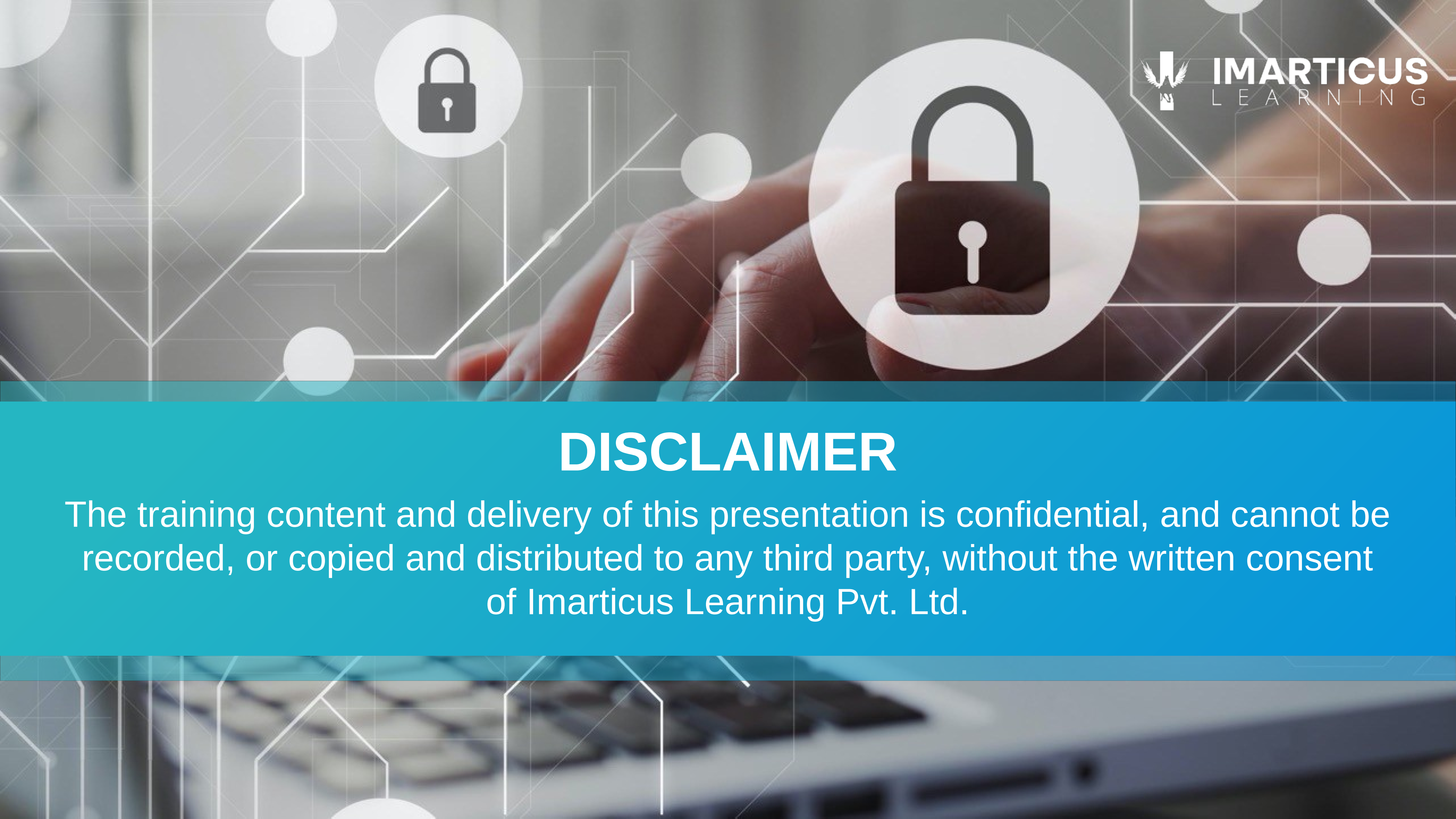
IMARTICUS LEARNING

Python Programming

NumPy



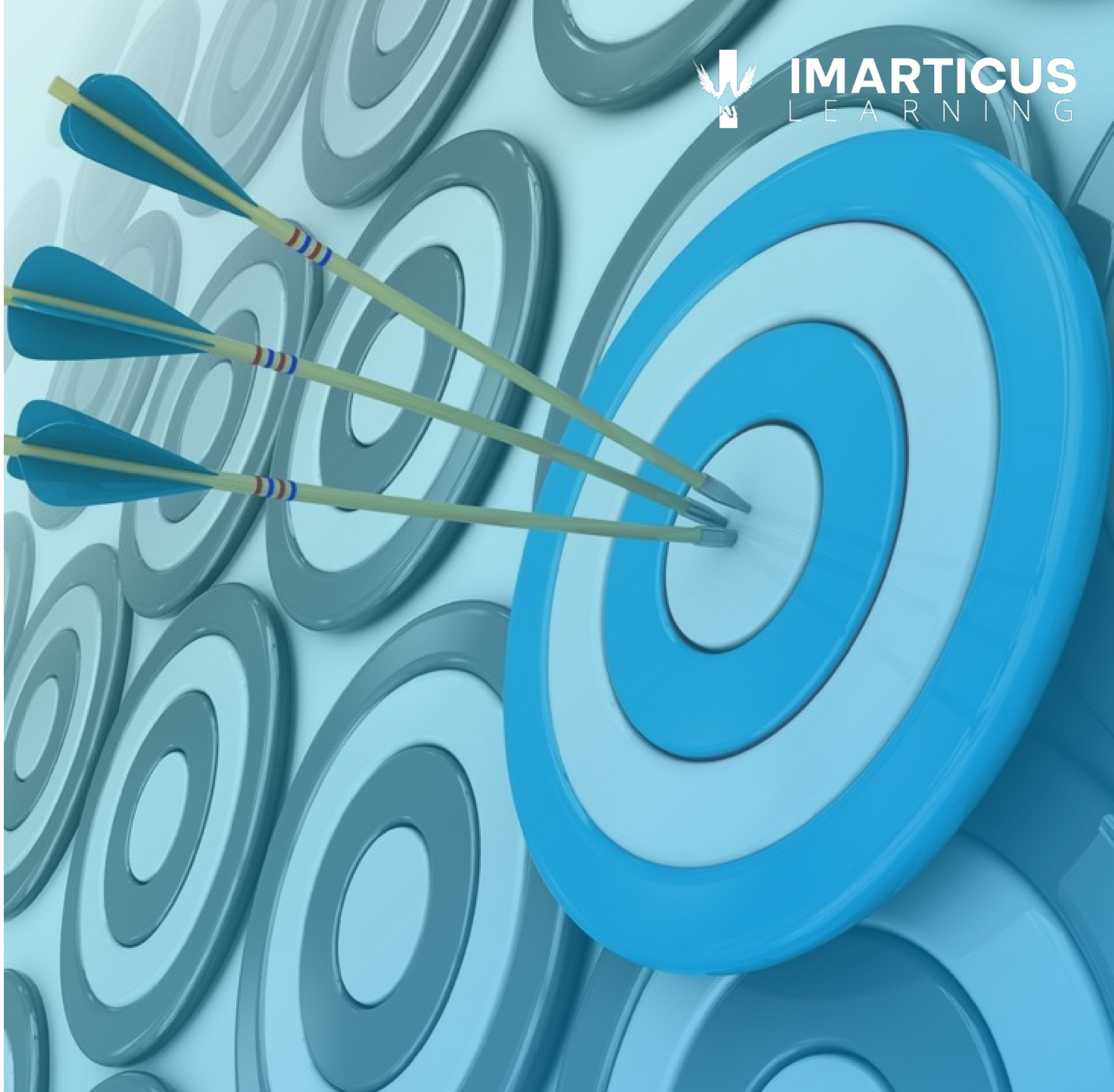
DISCLAIMER

The training content and delivery of this presentation is confidential, and cannot be recorded, or copied and distributed to any third party, without the written consent of Imarticus Learning Pvt. Ltd.

LEARNING OBJECTIVES

At the end of this session, you will learn:

- Introduction to NumPy
- NumPy Array
- Creating NumPy Array
- Array Attributes
- Array Methods
- Array Indexing
- Slicing Arrays
- Array Operation
- Iteration through Arrays

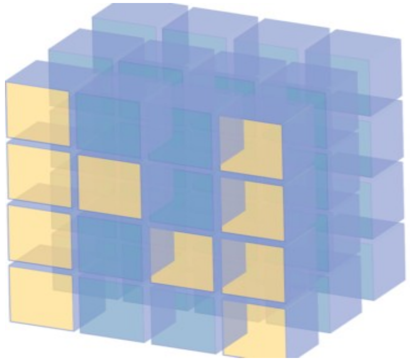


Introduction to NumPy

INTRODUCTION TO NUMPY

IMARTICUS LEARNING

NumPy stands for ‘Numeric Python’



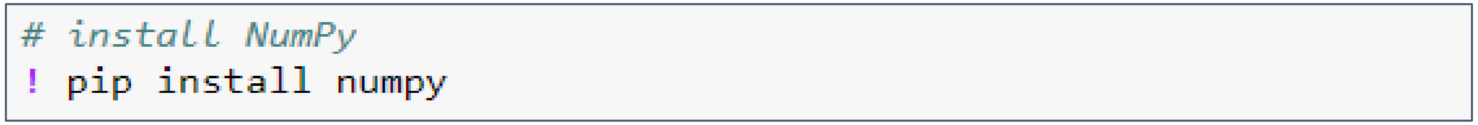
'Numerical Python’

- Used for mathematical and scientific computations
- Also provides ‘linalg’ module which contains functions like det, eig, norm to apply linear algebra on NumPy arrays
- NumPy array is the most widely used object of the NumPy library

INTRODUCTION TO NUMPY

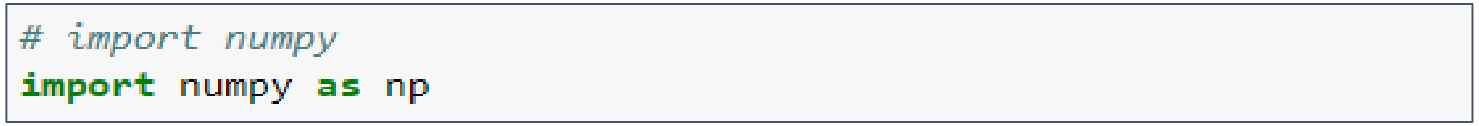
- Installing NumPy

Use the following command to install Numpy using jupyter notebook



```
# install NumPy
! pip install numpy
```

- Importing numpy as alias np is a common practice



```
# import numpy
import numpy as np
```

NumPy array

NUMPY ARRAY

Looks similar to a list

It is a grid of values, indexed by positive integers

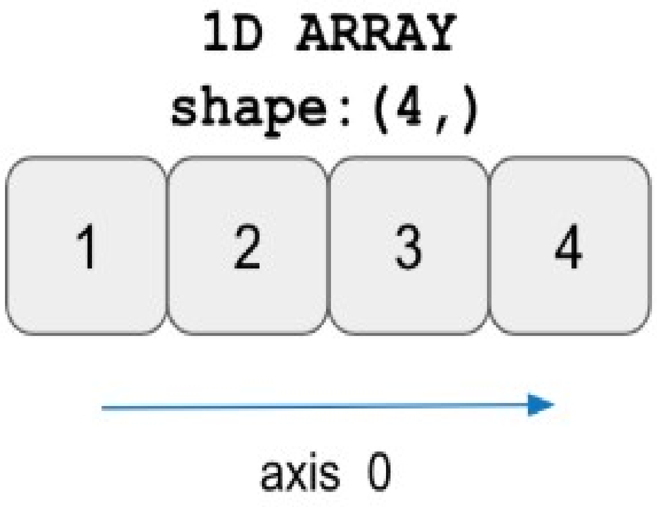
It generally contains numeric values. However it can also contain strings

Works faster than lists because it is homogeneous

It can be multidimensional

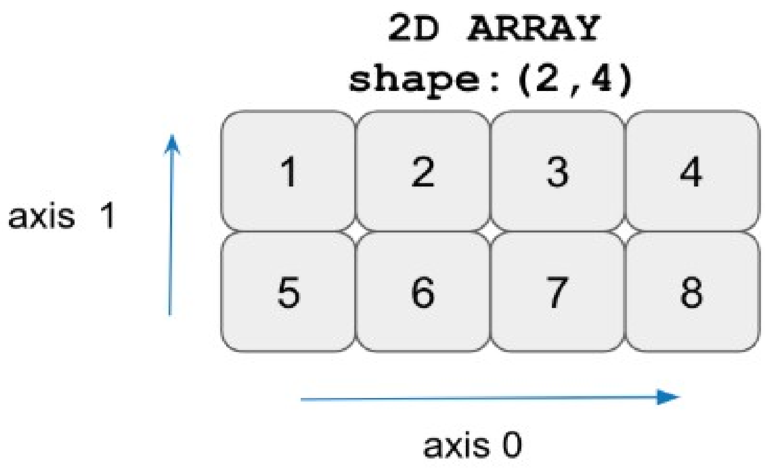
1D NUMPY ARRAY

- One dimensional array contains elements only in one dimension. In other words, the shape of the numpy array should contain only one value in the tuple



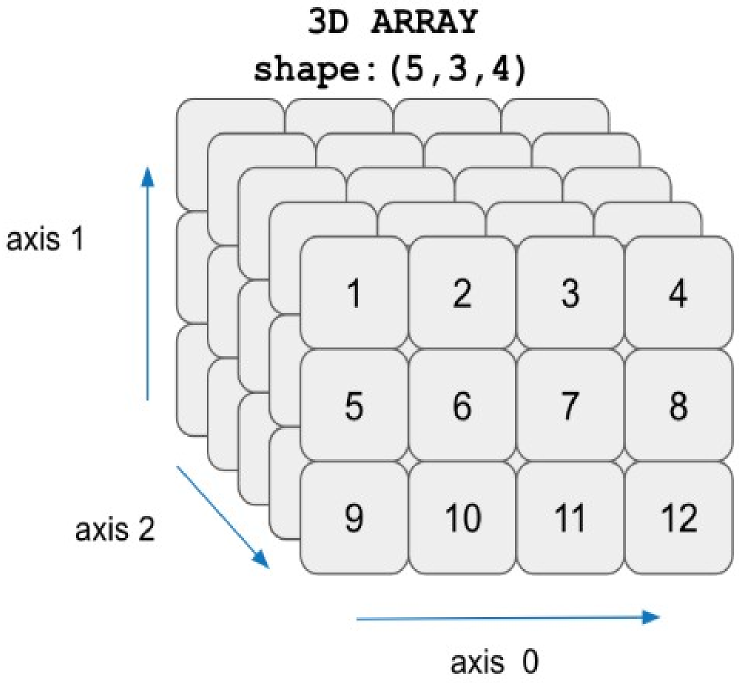
2D NUMPY ARRAY

- Two dimensional array is an array within an array
- The position of an data element is referred by two indices instead of one



3D NUMPY ARRAY

- A three-dimensional (3D) array is composed of 3 nested levels of arrays, one for each dimension

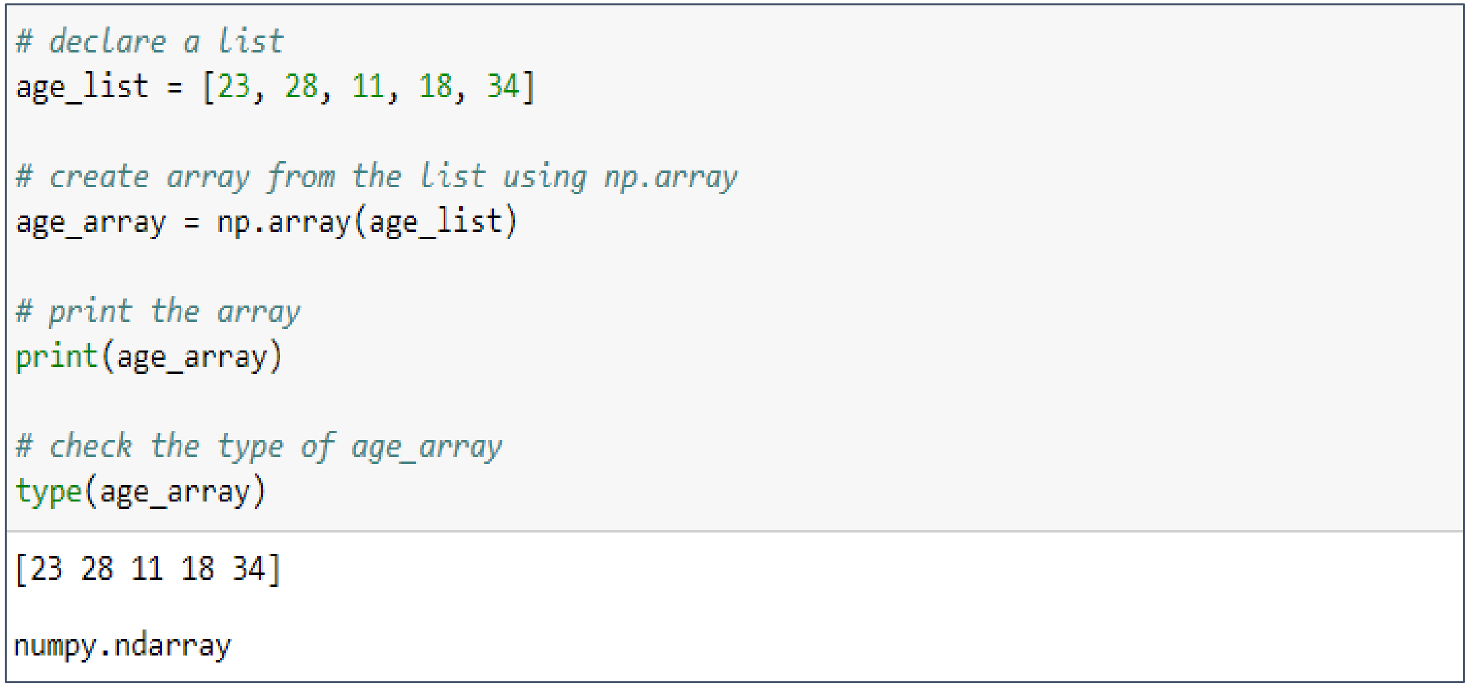


1	2	3	4
5	6	7	8
9	10	11	12

Creating NumPy array

CONVERTING A LIST INTO NUMPY ARRAY

- np.array() is used to create a numpy array from a list

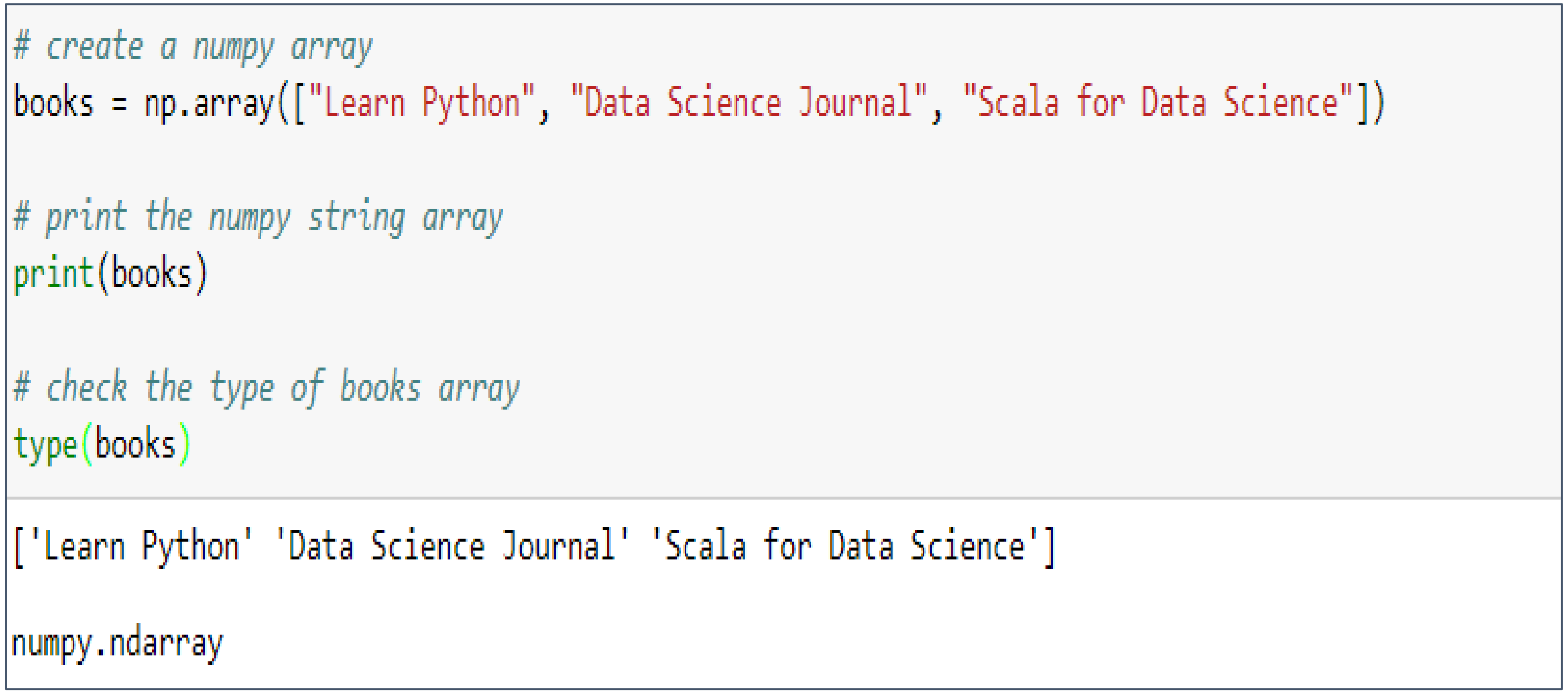


```
# declare a list
age_list = [23, 28, 11, 18, 34]

# create array from the list using np.array
age_array = np.array(age_list)

# print the array
print(age_array)

# check the type of age_array
type(age_array)
```

```
[23 28 11 18 34]

numpy.ndarray
```

CREATING NUMPY ARRAY

- Numpy arrays be used to create array of strings as well



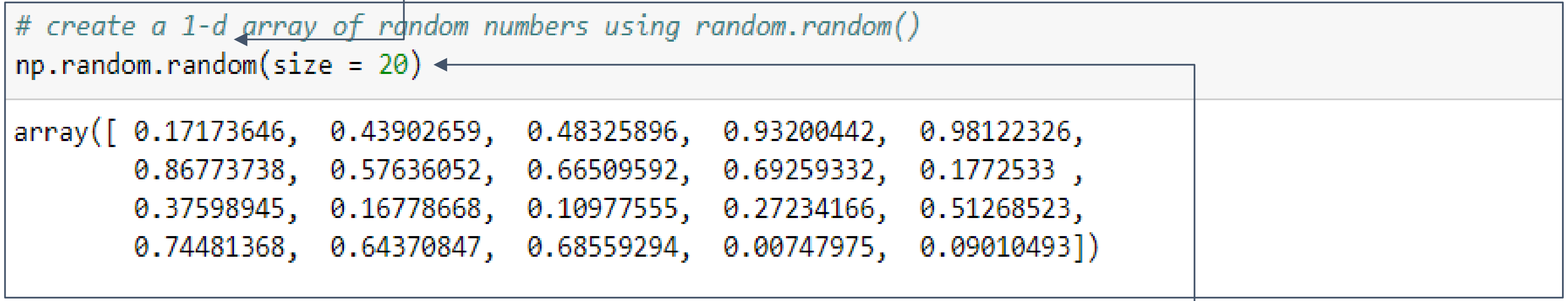
```
# create a numpy array
books = np.array(["Learn Python", "Data Science Journal", "Scala for Data Science"])

# print the numpy string array
print(books)

# check the type of books array
type(books)
```

```
['Learn Python' 'Data Science Journal' 'Scala for Data Science']

numpy.ndarray
```

NUMPY ARRAY OF RANDOM NUMBERS

- Create an array of 20 random numbers using random() method from the random module

random() method returns random numbers over the half-open interval [0.0, 1.0)



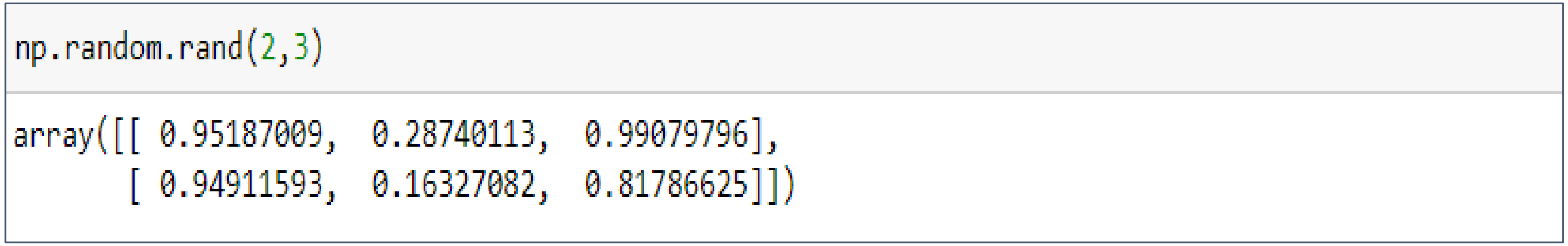
```
# create a 1-d array of random numbers using random.random()
np.random.random(size = 20)
```

```
array([ 0.17173646,  0.43902659,  0.48325896,  0.93200442,  0.98122326,
        0.86773738,  0.57636052,  0.66509592,  0.69259332,  0.1772533 ,
        0.37598945,  0.16778668,  0.10977555,  0.27234166,  0.51268523,
        0.74481368,  0.64370847,  0.68559294,  0.00747975,  0.09010493])
```

The required number of random numbers is passed through the ‘size’ parameter

NUMPY ARRAY OF RANDOM NUMBERS

- rand() method creates an array of random numbers of the given shape and between (0, 1)
- The dimensions of the returned array, should all be positive
- If no argument is given a single Python float is returned

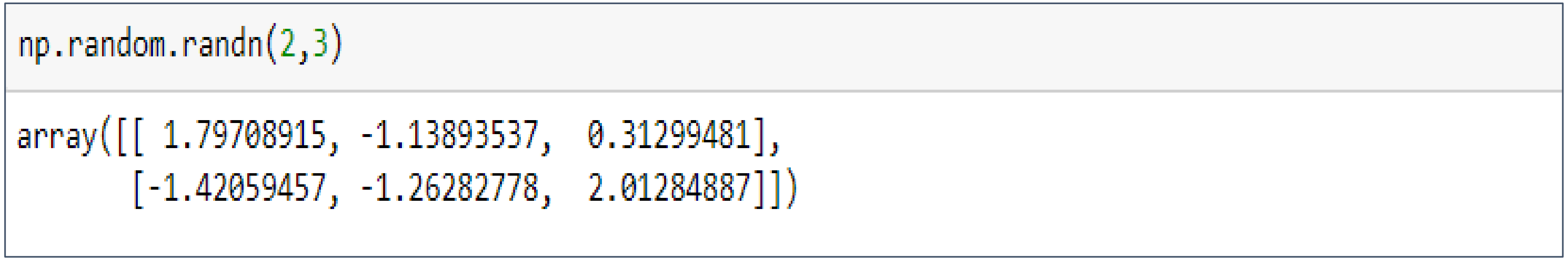


```
np.random.rand(2,3)
```

```
array([[ 0.95187009,  0.28740113,  0.99079796],
       [ 0.94911593,  0.16327082,  0.81786625]])
```

NUMPY ARRAY OF RANDOM NUMBERS

- The randn() returns a set of values from the standard normal distribution
- The dimensions of the returned array, should all be positive
- If no argument is given a single Python float is returned

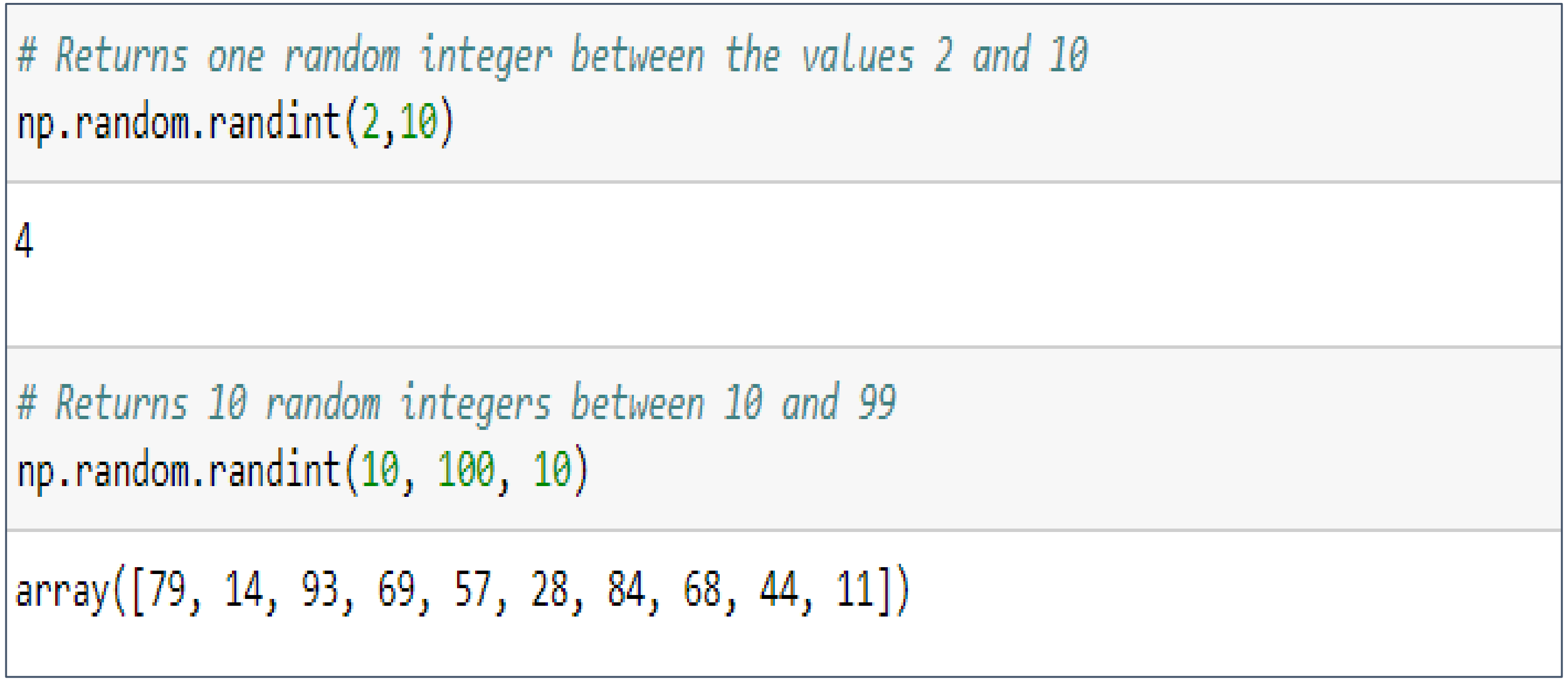


```
np.random.randn(2,3)
```

```
array([[ 1.79708915, -1.13893537,  0.31299481],
       [-1.42059457, -1.26282778,  2.01284887]])
```

NUMPY ARRAY OF RANDOM NUMBERS

- The randint() returns random integers from low (inclusive) to high (exclusive)



```
# Returns one random integer between the values 2 and 10
np.random.randint(2,10)
```

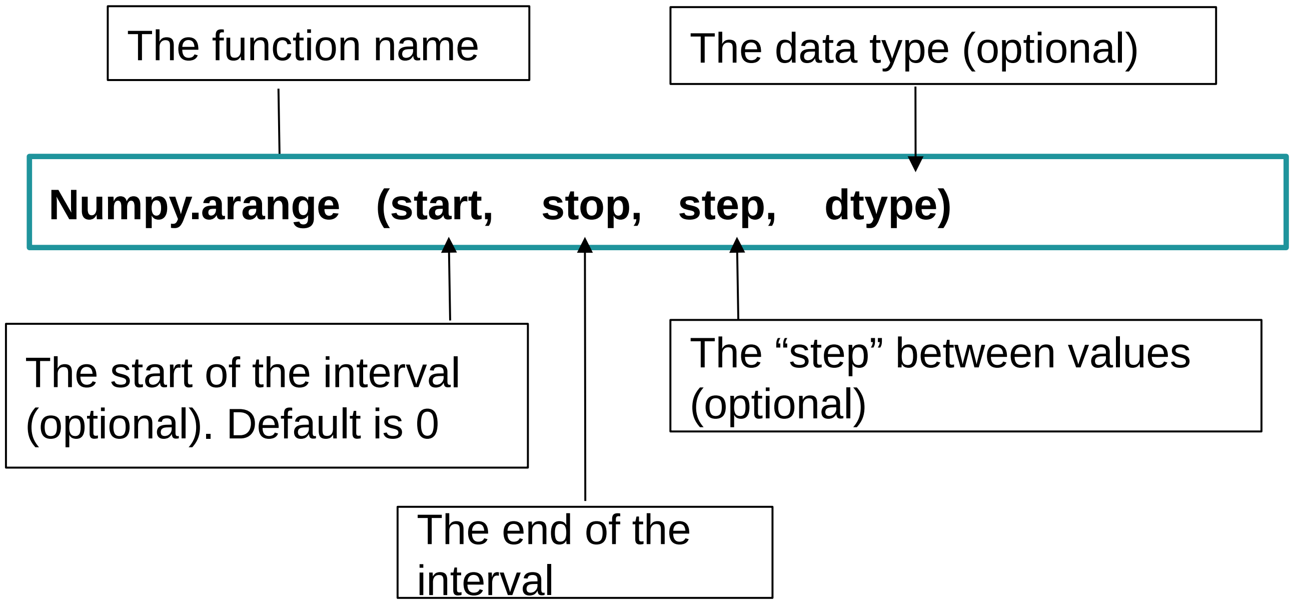
```
4
```

```
# Returns 10 random integers between 10 and 99
np.random.randint(10, 100, 10)
```

```
array([79, 14, 93, 69, 57, 28, 84, 68, 44, 11])
```

CREATING NUMPY ARRAY USING ARANGE()

- np.arange() can also be used to create a NumPy array
- The numbers generated have the same difference
- The function generates as many possible numbers in the given range



The function name

The data type (optional)

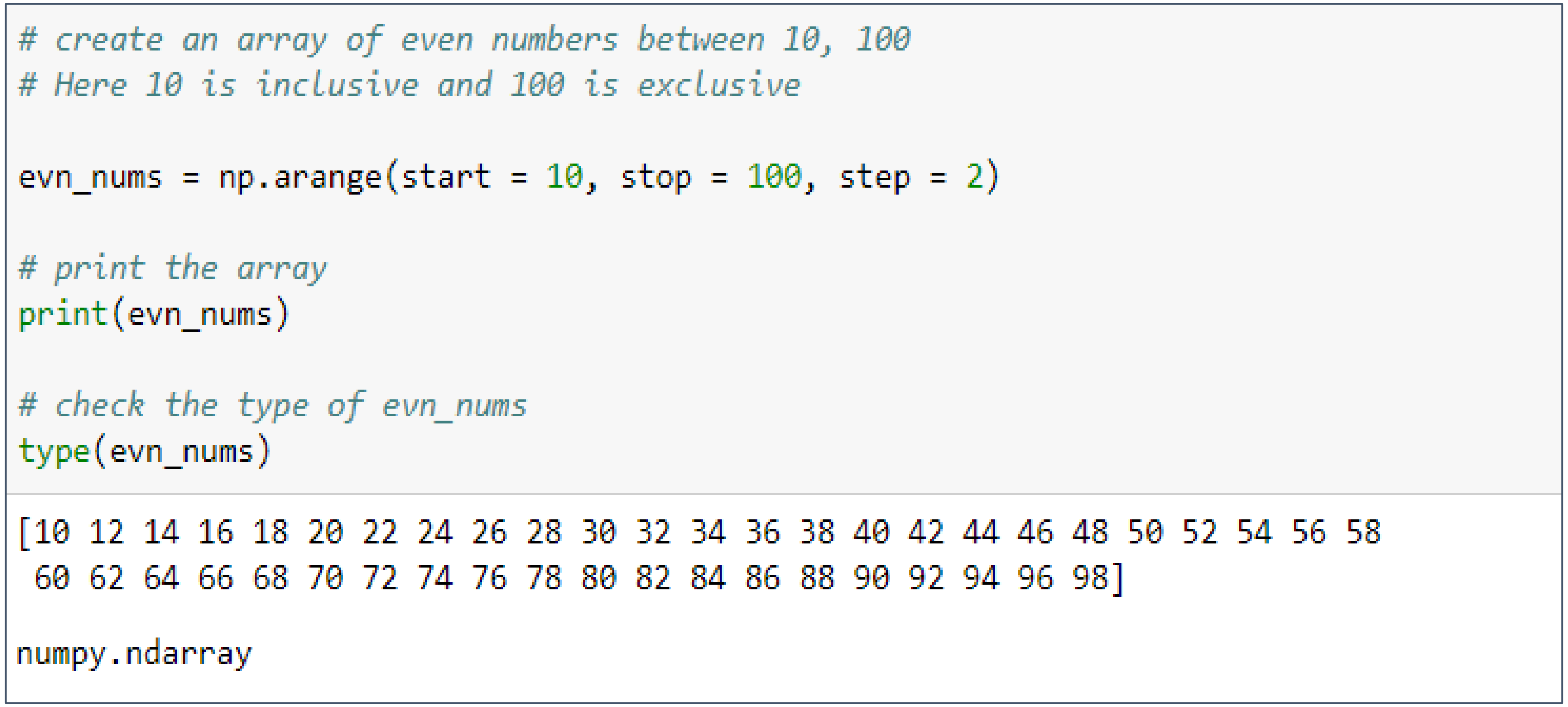
Numpy.arange (start, stop, step, dtype)

The start of the interval (optional). Default is 0

The end of the interval

The “step” between values (optional)

CREATING NUMPY ARRAY USING ARANGE()



```python
# create an array of even numbers between 10, 100
# Here 10 is inclusive and 100 is exclusive

evn_nums = np.arange(start = 10, stop = 100, step = 2)

# print the array
print(evn_nums)

# check the type of evn_nums
type(evn_nums)
```

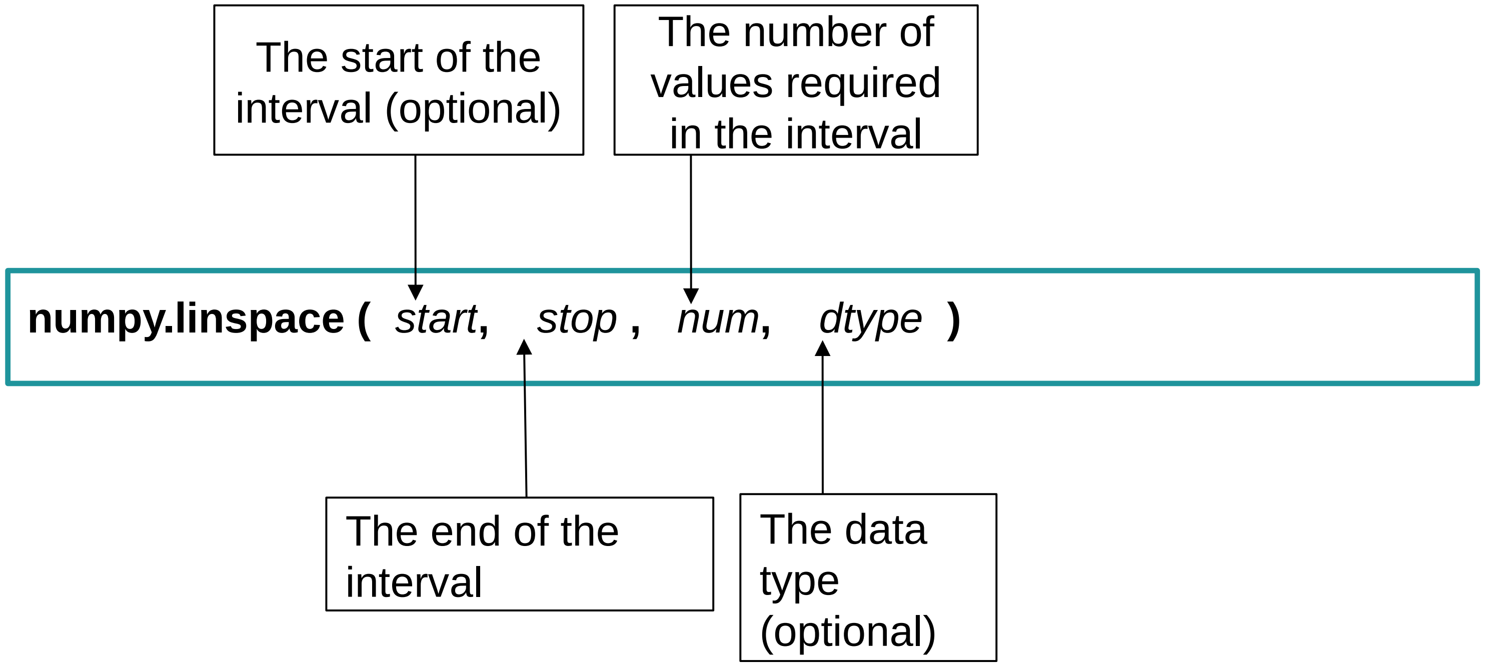
```
[10 12 14 16 18 20 22 24 26 28 30 32 34 36 38 40 42 44 46 48 50 52 54 56 58
 60 62 64 66 68 70 72 74 76 78 80 82 84 86 88 90 92 94 96 98]

numpy.ndarray
```

- The np.arange() create a series of values from 10 to 100 with a difference of 2, stored as a numpy array

CREATING NUMPY ARRAY USING LINSPACE()

- linspace() generates a specified number of values in a specified range

Syntax:



numpy.linspace (*start*, *stop* **,** *num*, *dtype* **)**

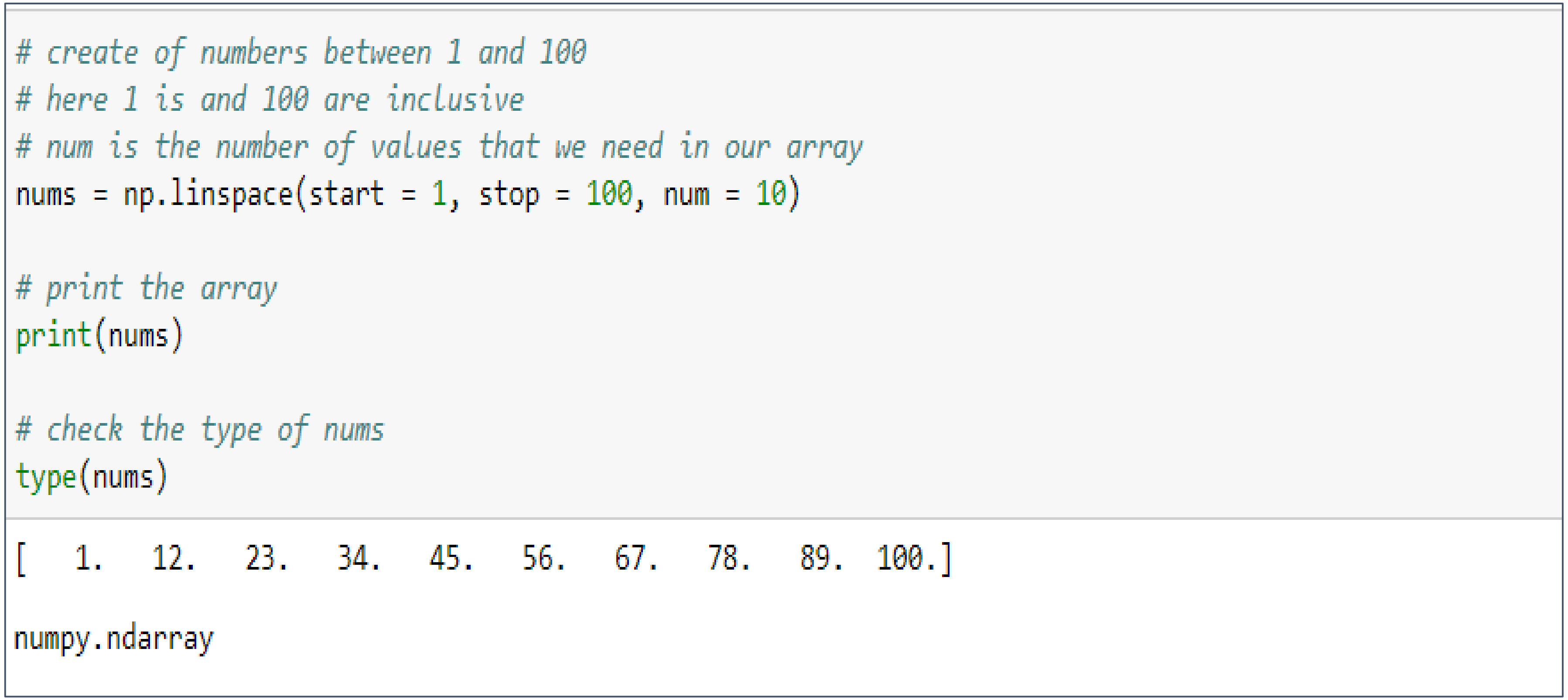
The start of the interval (optional)

The number of values required in the interval

The end of the interval

The data type (optional)

CREATING NUMPY ARRAY USING LINSPACE()



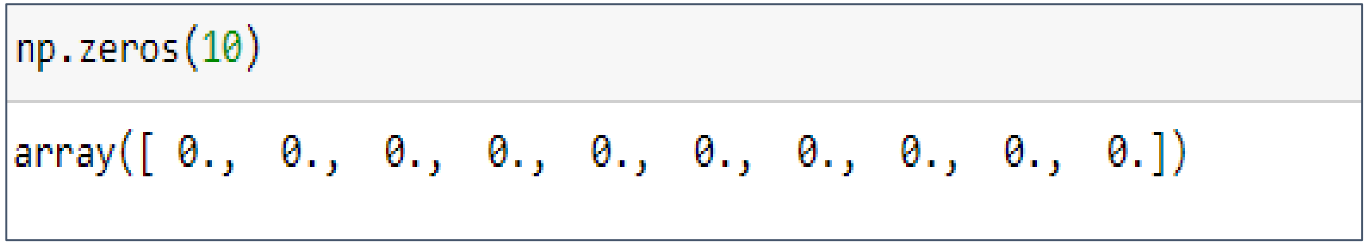
```
# create of numbers between 1 and 100
# here 1 is and 100 are inclusive
# num is the number of values that we need in our array
nums = np.linspace(start = 1, stop = 100, num = 10)

# print the array
print(nums)

# check the type of nums
type(nums)
```

```
[  1.  12.  23.  34.  45.  56.  67.  78.  89. 100.]

numpy.ndarray
```

- np.linspace() produces a sequence of 10 evenly spaced values from 1 to 100, stored as a numpy array

CREATING NUMPY ARRAY OF ZEROES

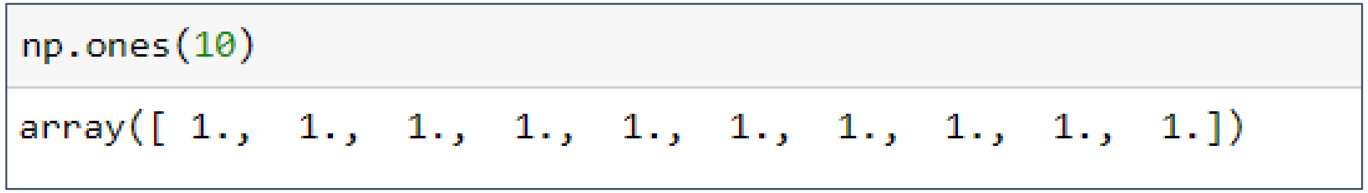
- Creating 1D numpy array of zeros



```
np.zeros(10)
```

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

- Creating 1D numpy array of ones



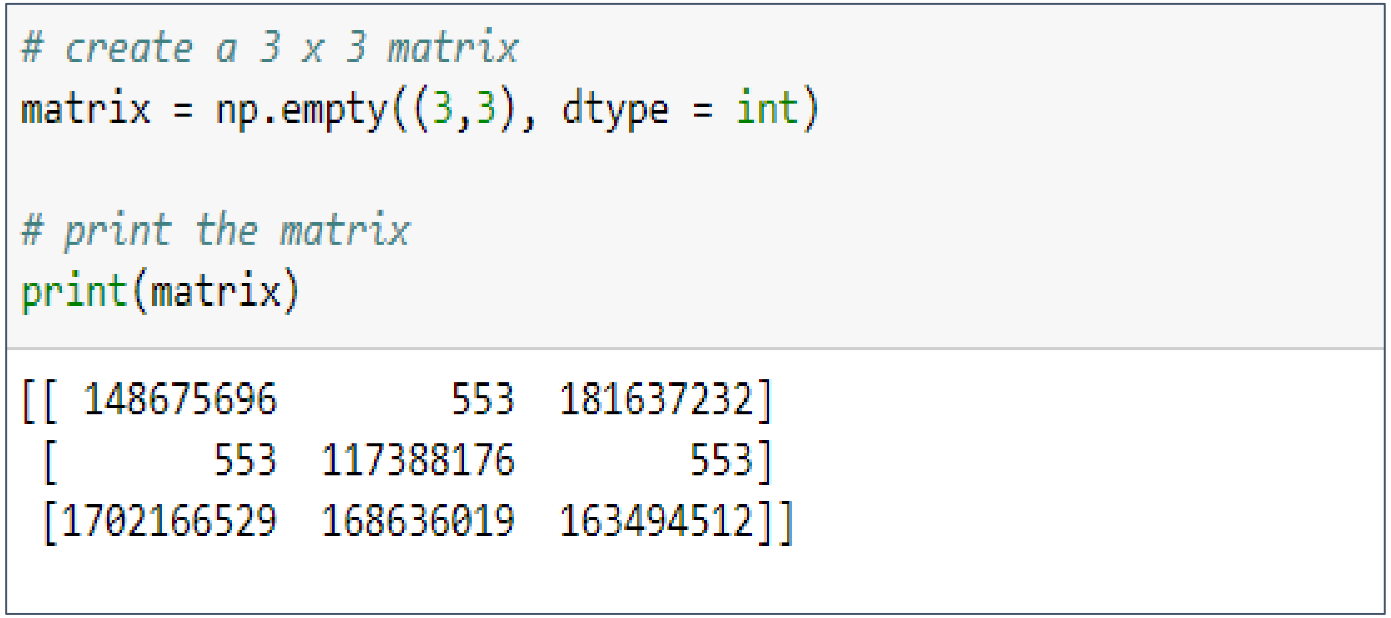
```
np.ones(10)
```

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

CREATING 2D NUMPY ARRAY

- np.empty() returns the matrix with arbitrary values of given shape and data type

- ‘dtype = object’ returns None values

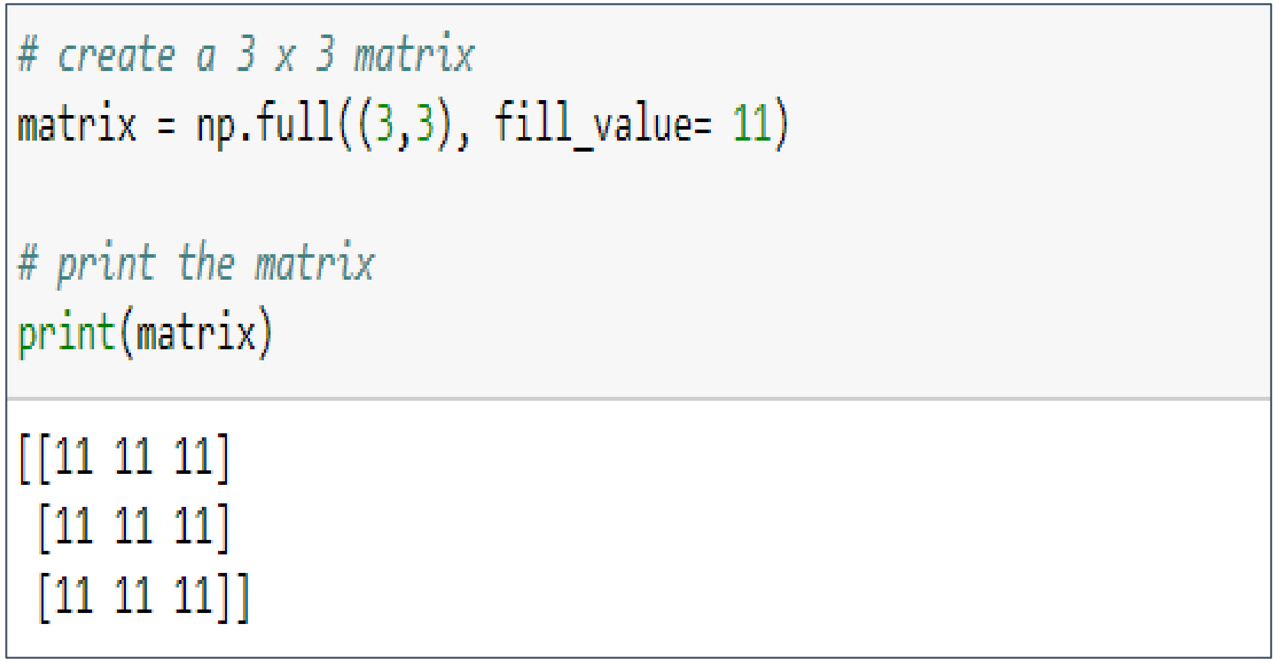


```
# create a 3 x 3 matrix
matrix = np.empty((3,3), dtype = int)

# print the matrix
print(matrix)
```

```
[[ 148675696        553  181637232]
 [       553  117388176        553]
 [1702166529  168636019  163494512]]
```

CREATING 2D NUMPY ARRAY

- np.full() returns the matrix of given shape with the value set by the ‘fill_value’ parameter

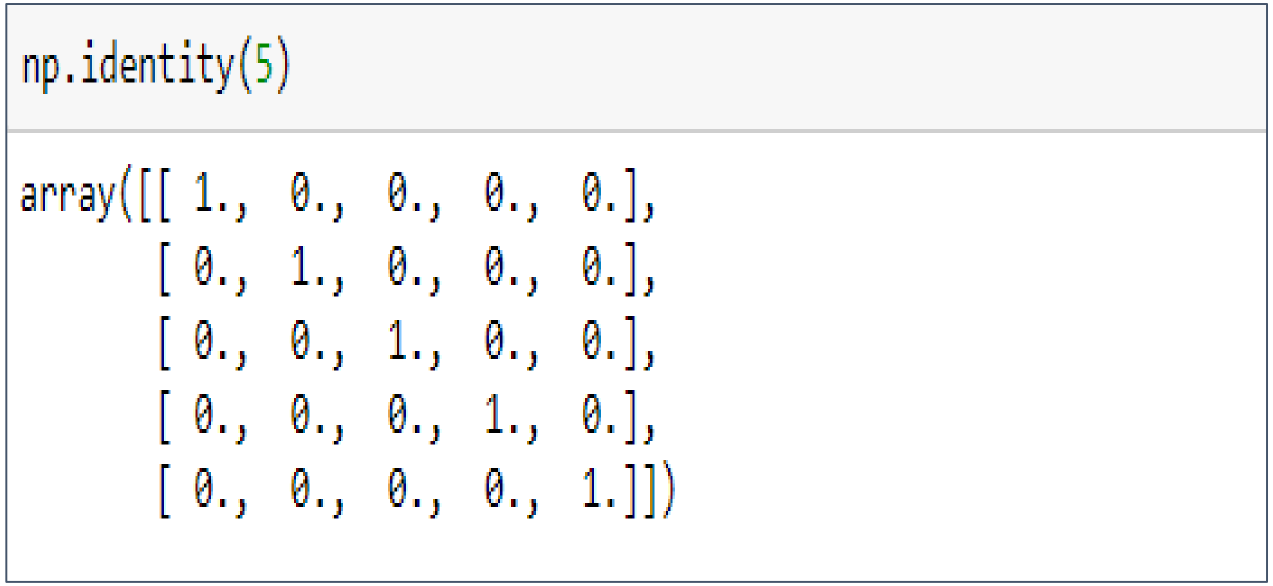


```
# create a 3 x 3 matrix
matrix = np.full((3,3), fill_value= 11)

# print the matrix
print(matrix)
```

```
[[11 11 11]
 [11 11 11]
 [11 11 11]]
```

CREATING 2D NUMPY ARRAY

- np.identity() returns the identity matrix of specified shape



```
np.identity(5)
```

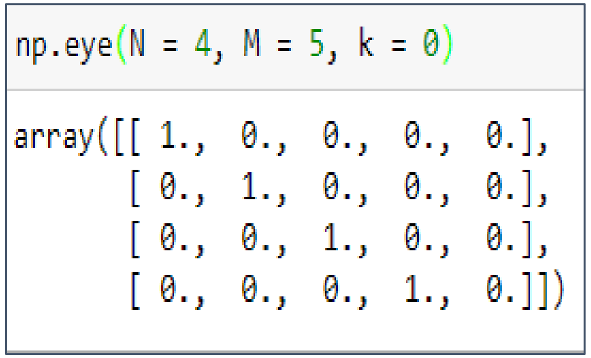
```
array([[ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.],
       [ 0.,  0.,  1.,  0.,  0.],
       [ 0.,  0.,  0.,  1.,  0.],
       [ 0.,  0.,  0.,  0.,  1.]])
```

CREATING 2D NUMPY ARRAY

- np.eye() creates NxM matrix with value ‘1’ on the k-th diagonal and remaining entries as zero

- K > 0 represents upper diagonal
- K < 0 represents lower diagonal

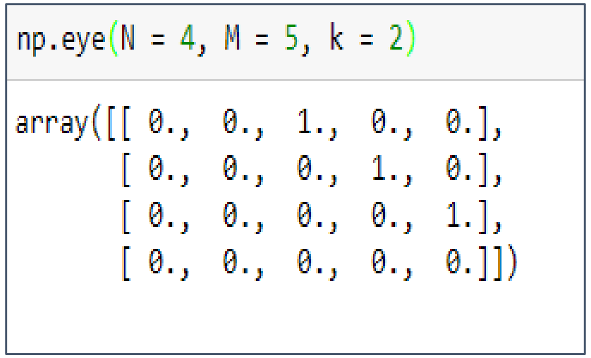
K = 0 represents main diagonal



```
np.eye(N = 4, M = 5, k = 0)
```

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

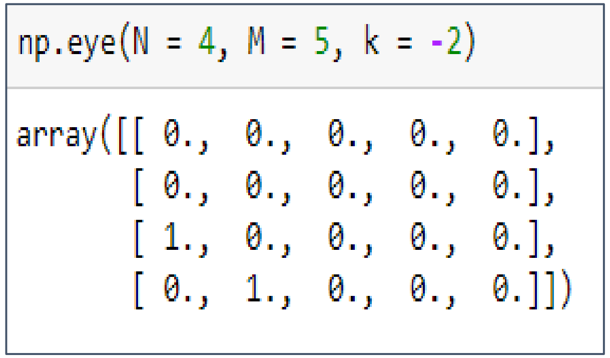
K > 0 represents upper diagonal



```
np.eye(N = 4, M = 5, k = 2)
```

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

K < 0 represents lower diagonal



```
np.eye(N = 4, M = 5, k = -2)
```

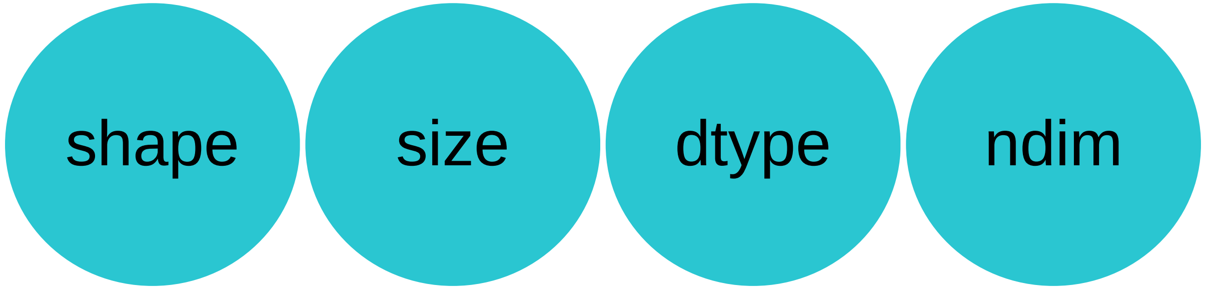
```
array([[ 0.,  0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.,  0.],
       [ 1.,  0.,  0.,  0.,  0.],
       [ 0.,  1.,  0.,  0.,  0.]])
```

Array Attributes

ARRAY ATTRIBUTES

Attributes are the features/characteristics of an object that describes the object

Some of the attributes of the numpy array are:



shape

size

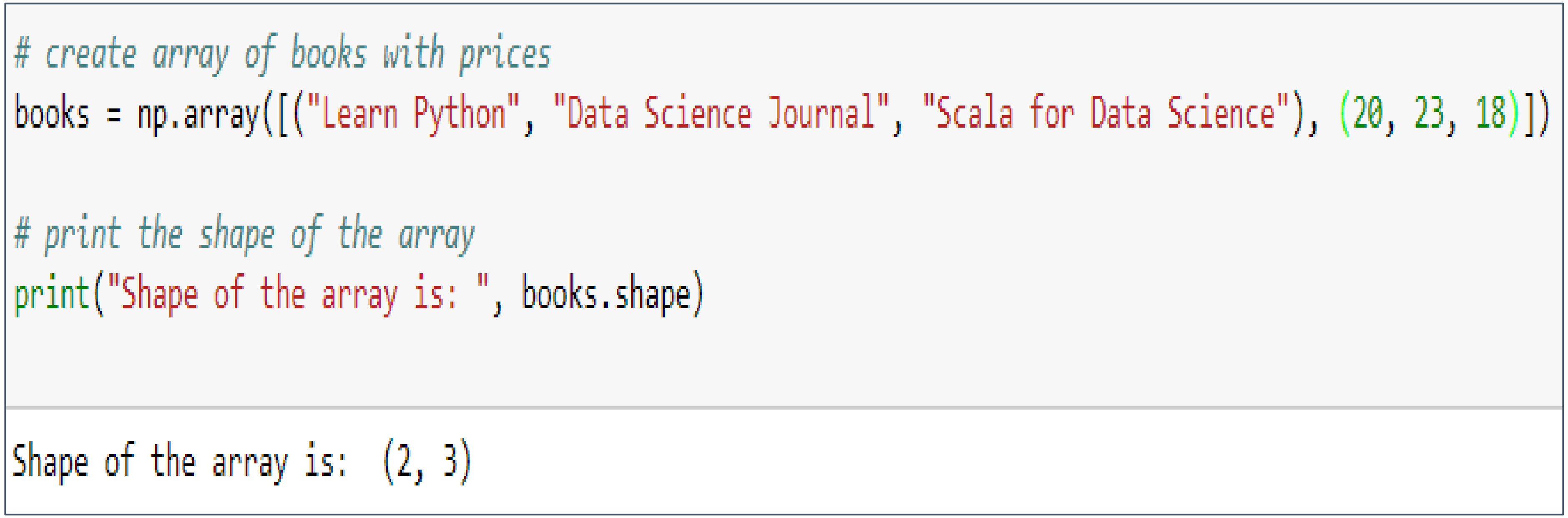
dtype

ndim

Attributes do not have parentheses following them

ARRAY ATTRIBUTES-SHAPE

- The shape returns the number of rows and columns of the array respectively

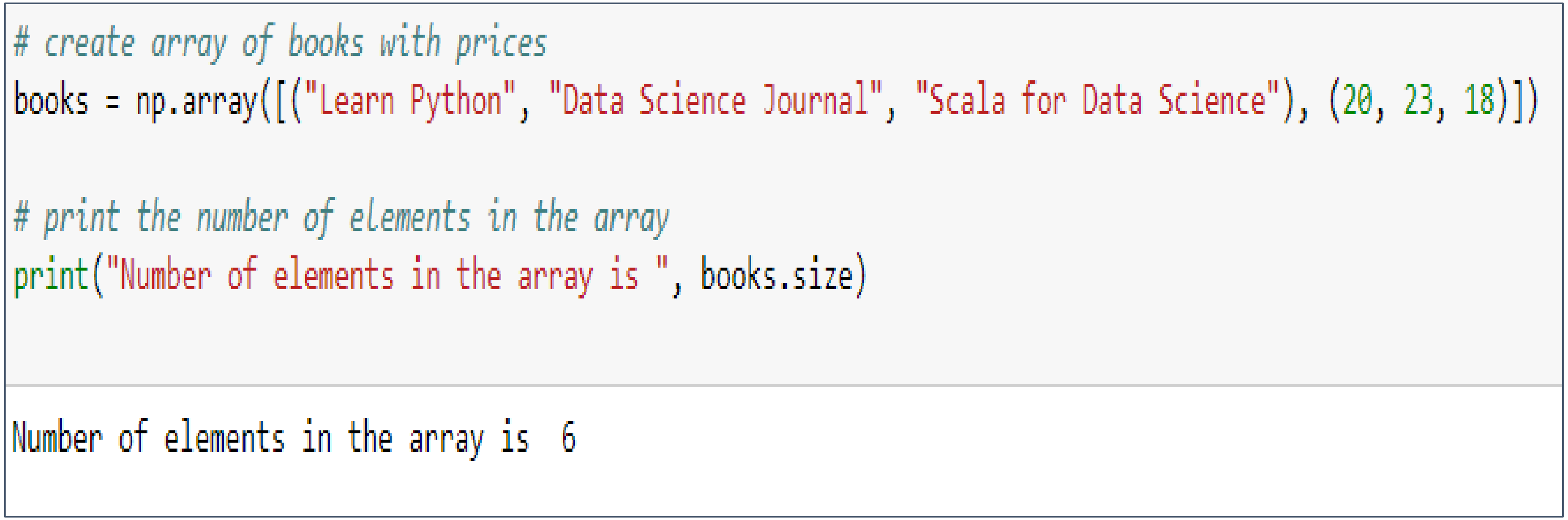


```
# create array of books with prices
books = np.array([("Learn Python", "Data Science Journal", "Scala for Data Science"), (20, 23, 18)])

# print the shape of the array
print("Shape of the array is: ", books.shape)
```

```
Shape of the array is:  (2, 3)
```

ARRAY ATTRIBUTES-SIZE

- The size returns the number of elements in an array

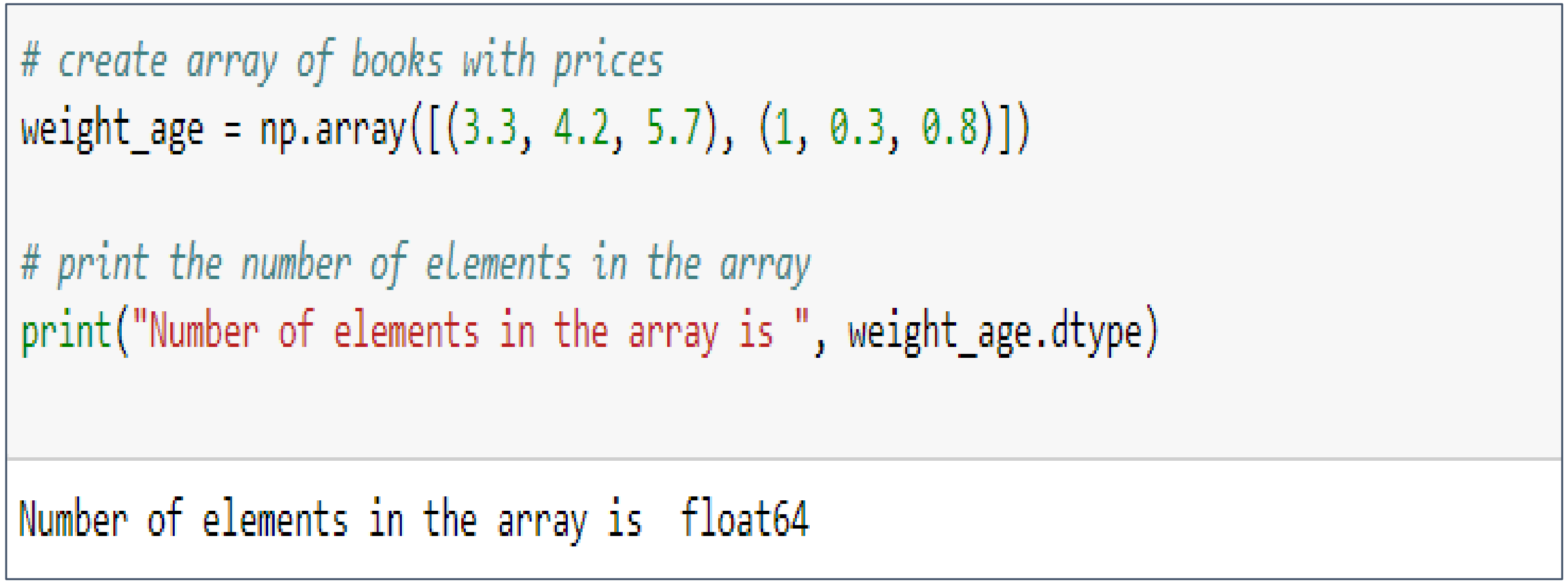


```
# create array of books with prices
books = np.array([("Learn Python", "Data Science Journal", "Scala for Data Science"), (20, 23, 18)])

# print the number of elements in the array
print("Number of elements in the array is ", books.size)
```

```
Number of elements in the array is  6
```

ARRAY ATTRIBUTES-DTYPE

- The dtype returns the type of the data along with the size in bytes



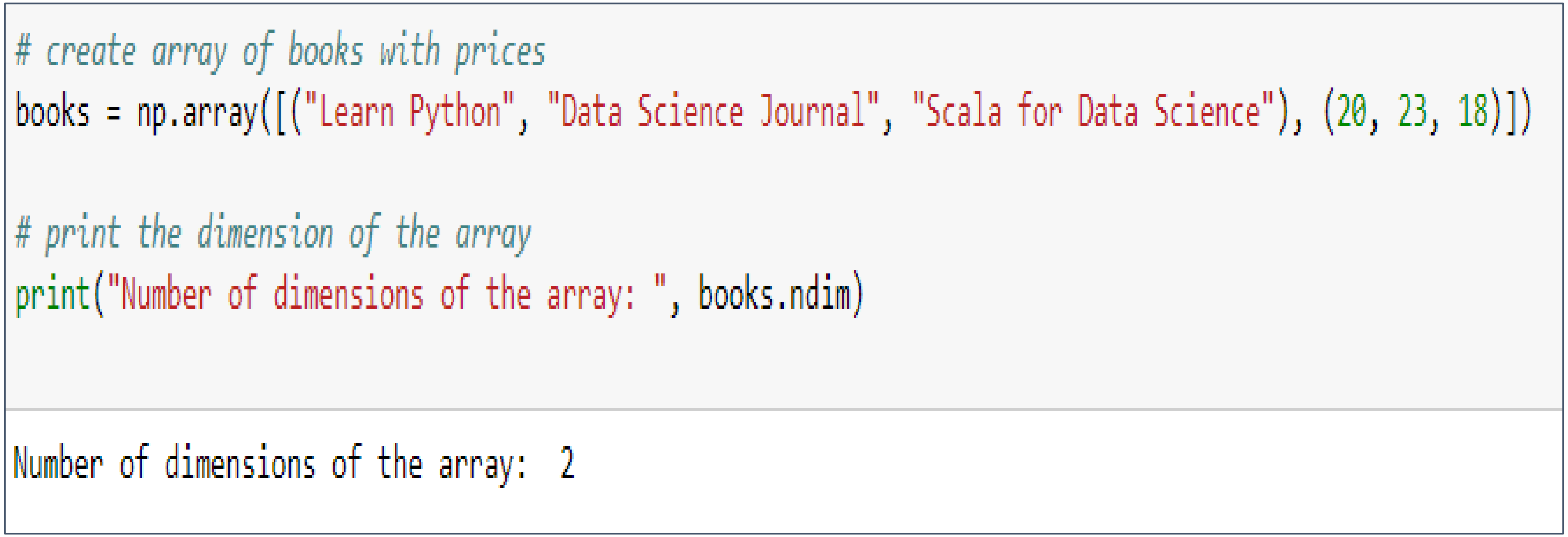
```
# create array of books with prices
weight_age = np.array([(3.3, 4.2, 5.7), (1, 0.3, 0.8)])

# print the number of elements in the array
print("Number of elements in the array is ", weight_age.dtype)
```

```
Number of elements in the array is  float64
```

In this example, the array consists of 64-bit floating-point numbers. Thus, the dtype of the array is float64

ARRAY ATTRIBUTES-NDIM

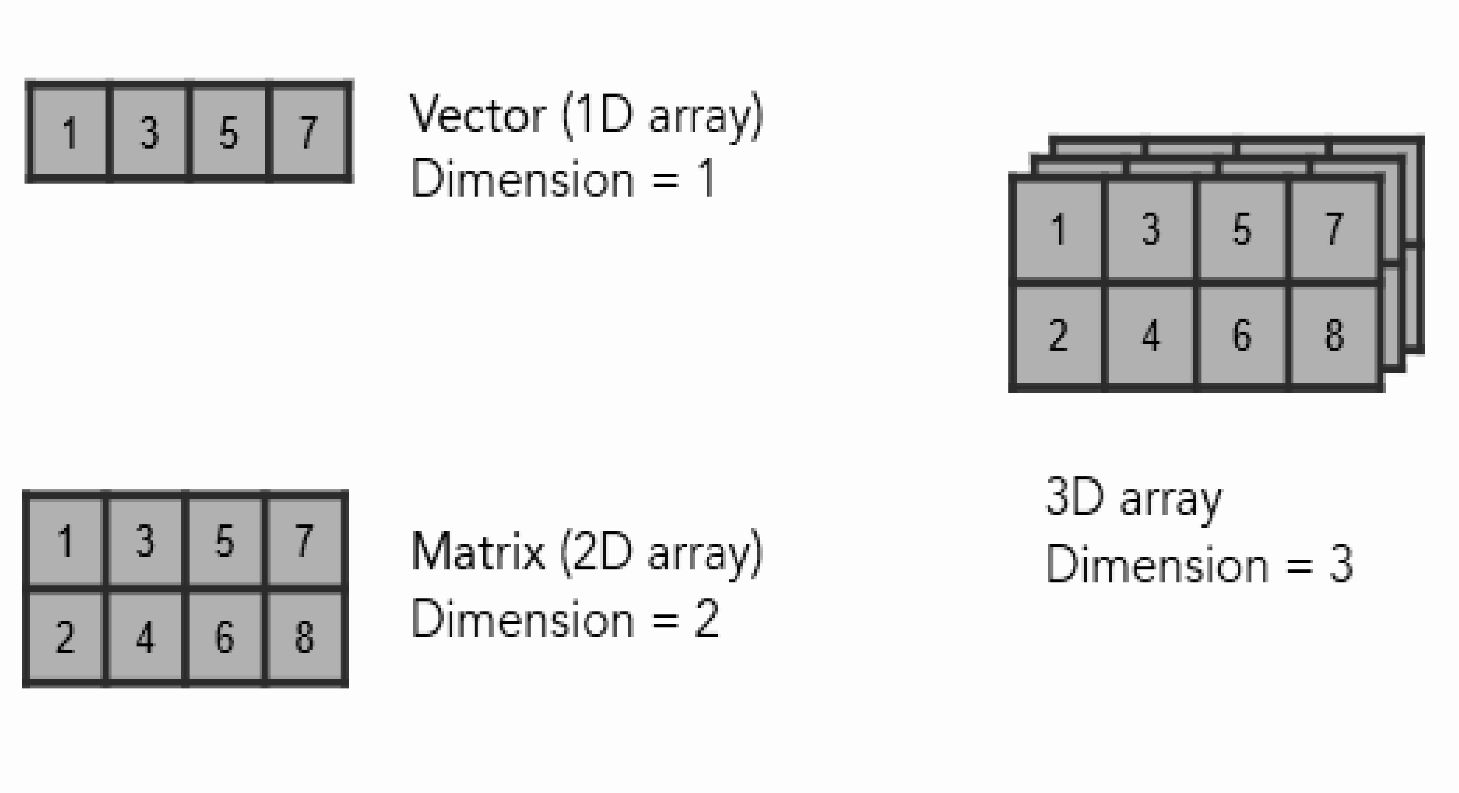
- The ndim returns the number of axes (dimension) of the array



```
# create array of books with prices
books = np.array([("Learn Python", "Data Science Journal", "Scala for Data Science"), (20, 23, 18)])

# print the dimension of the array
print("Number of dimensions of the array: ", books.ndim)
```

```
Number of dimensions of the array:  2
```

ARRAY ATTRIBUTES-NDIM



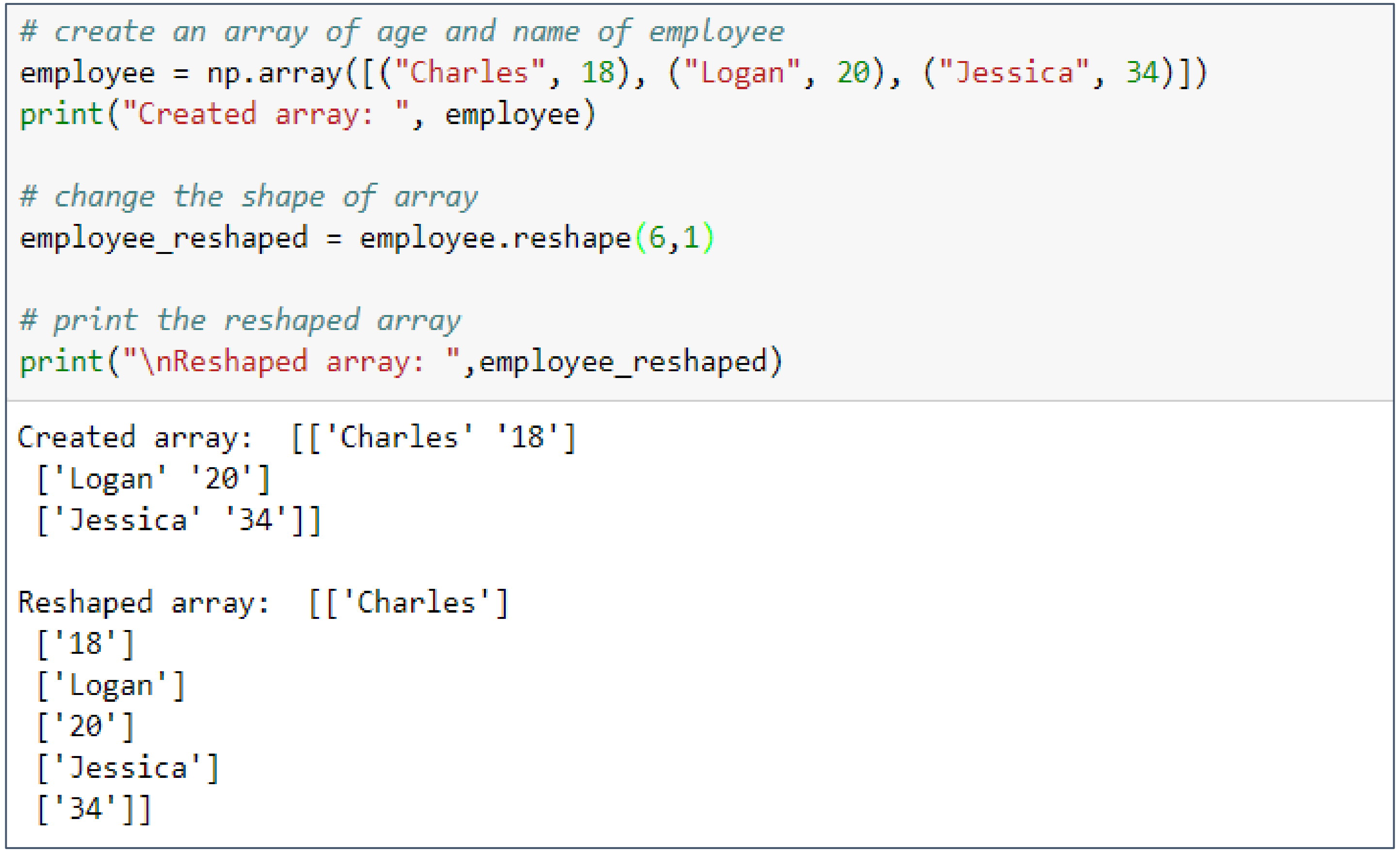
Array Methods

ARRAY METHODS

Methods are object functions that takes parameters in the parentheses and returns the modified object

ARRAY METHODS

- In this example, the reshape(6, 1) is reshaping a 3 X 2 array to 6 X 1 array



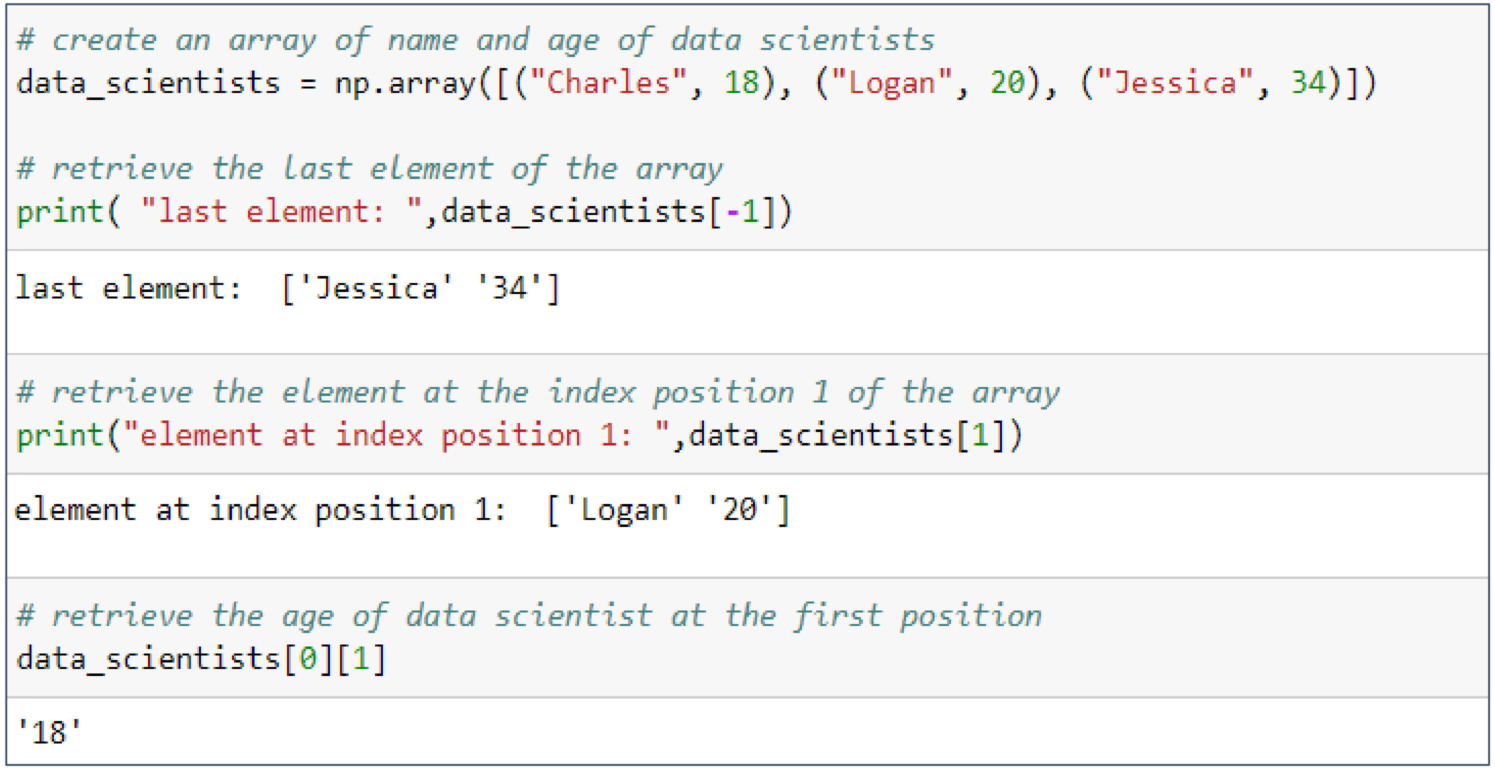
```python
# create an array of age and name of employee
employee = np.array([("Charles", 18), ("Logan", 20), ("Jessica", 34)])
print("Created array: ", employee)

# change the shape of array
employee_reshaped = employee.reshape(6,1)

# print the reshaped array
print("\nReshaped array: ",employee_reshaped)
```

```
Created array:  [['Charles' '18']
 ['Logan' '20']
 ['Jessica' '34']]

Reshaped array:  [['Charles']
 ['18']
 ['Logan']
 ['20']
 ['Jessica']
 ['34']]
```

Array Indexing

INDEXING ARRAY

- The element in the array can be accessed by the positional index of the element
- The index for an array starts at 0 from left and at -1 starts from the right



```
# create an array of name and age of data scientists
data_scientists = np.array([("Charles", 18), ("Logan", 20), ("Jessica", 34)])

# retrieve the last element of the array
print( "last element: ",data_scientists[-1])
```

```
last element:  ['Jessica' '34']
```

```
# retrieve the element at the index position 1 of the array
print("element at index position 1: ",data_scientists[1])
```

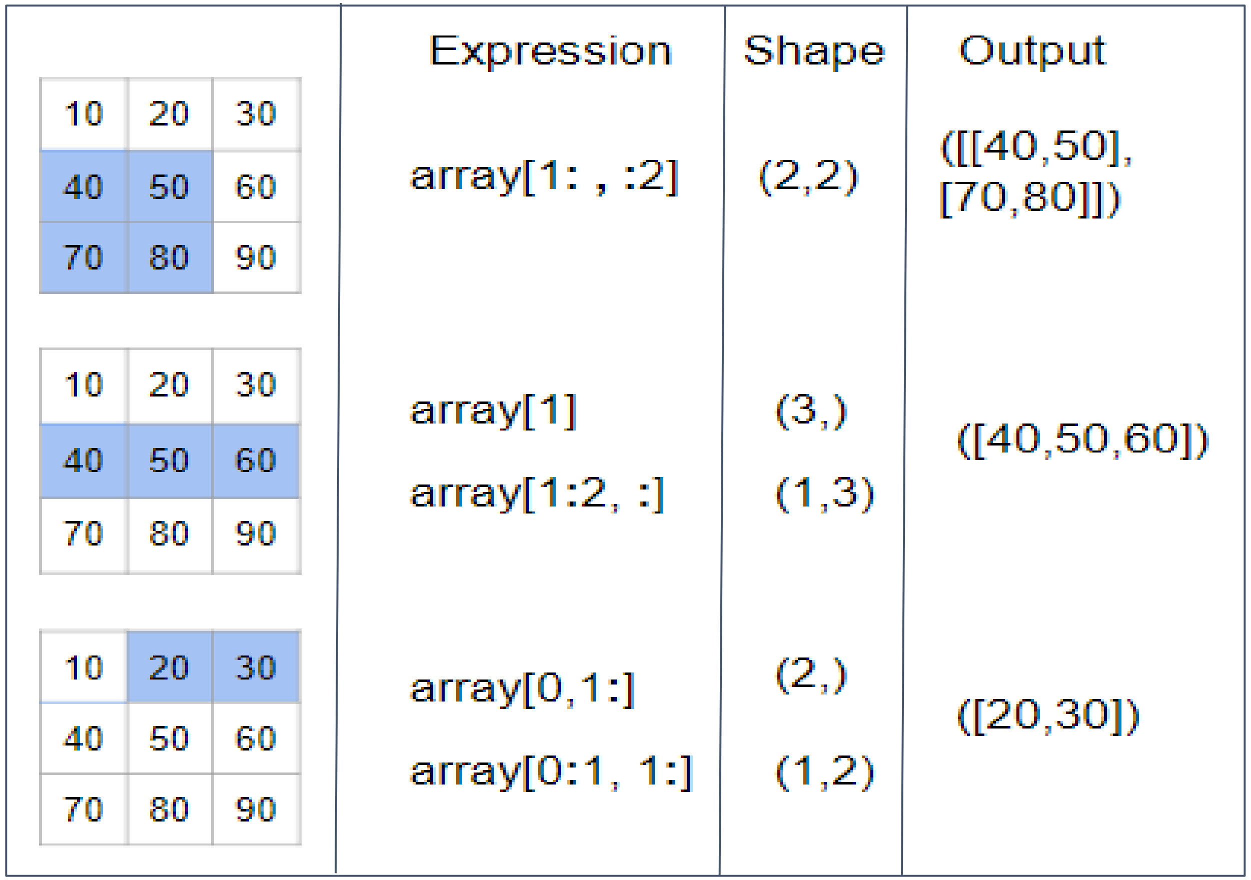
```
element at index position 1:  ['Logan' '20']
```

```
# retrieve the age of data scientist at the first position
data_scientists[0][1]
```

```
'18'
```

Slicing Array

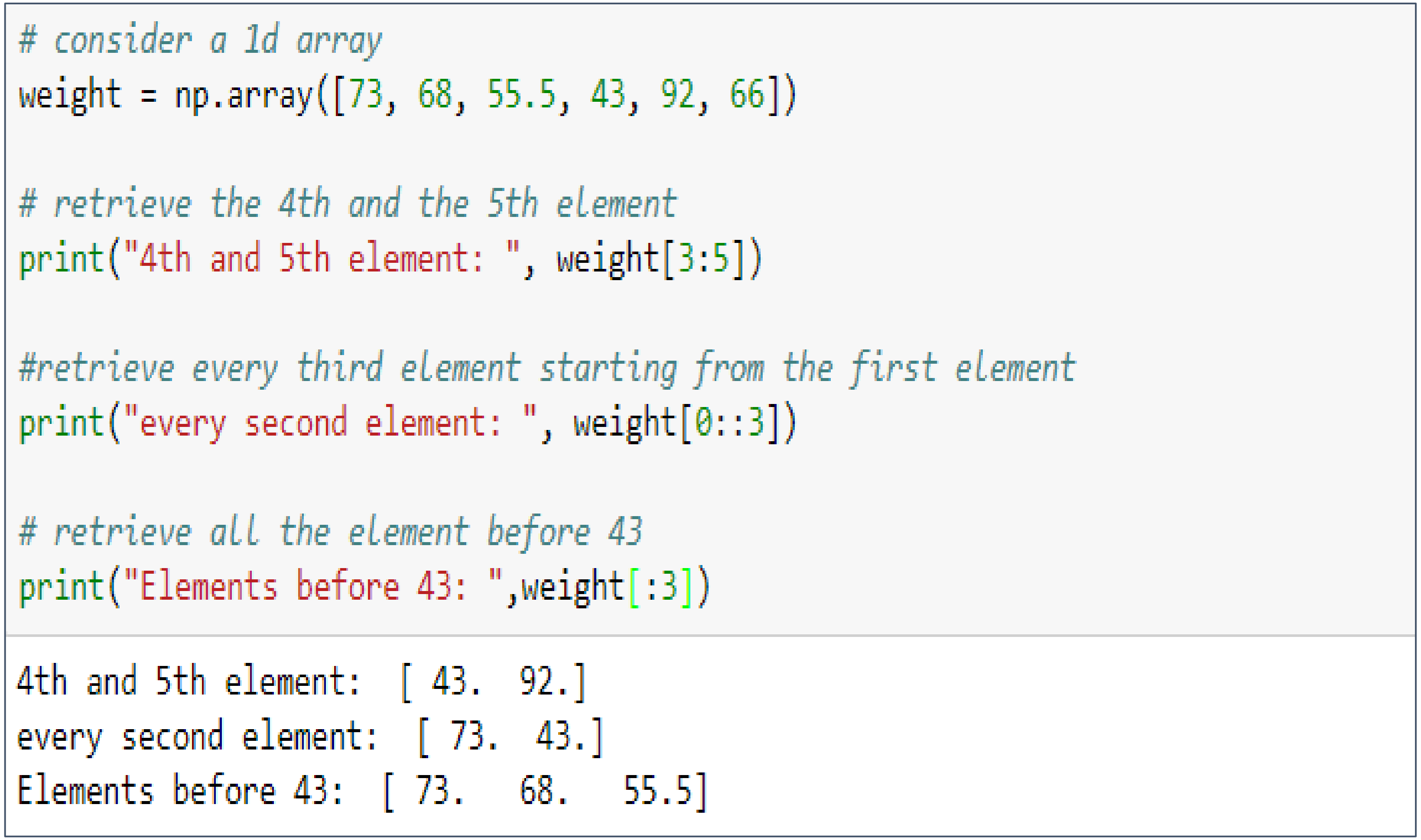
SLICING ARRAYS



	Expression	Shape	Output
10 20 30 40 50 60 70 80 90	array[1: , :2]	(2,2)	([[40,50], [70,80]])
10 20 30 40 50 60 70 80 90	array[1] array[1:2, :]	(3,) (1,3)	([40,50,60])
10 20 30 40 50 60 70 80 90	array[0,1:] array[0:1, 1:]	(2,) (1,2)	([20,30])

SLICING 1D ARRAY

- Slicing allows us to access more than one element

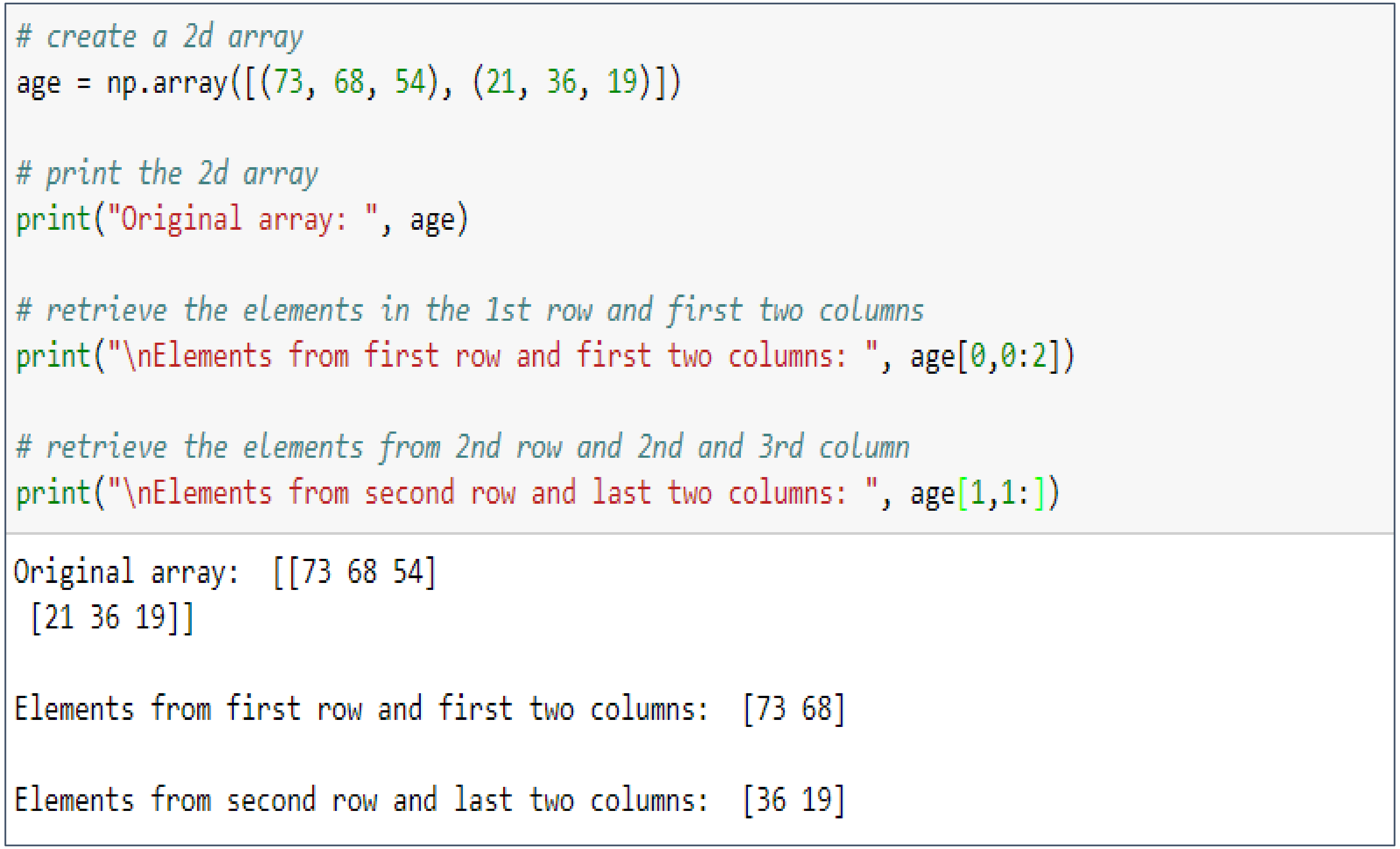


```
# consider a 1d array
weight = np.array([73, 68, 55.5, 43, 92, 66])

# retrieve the 4th and the 5th element
print("4th and 5th element: ", weight[3:5])

#retrieve every third element starting from the first element
print("every second element: ", weight[0::3])

# retrieve all the element before 43
print("Elements before 43: ",weight[:3])
```

```
4th and 5th element:  [ 43.  92.]
every second element:  [ 73.  43.]
Elements before 43:  [ 73.   68.   55.5]
```

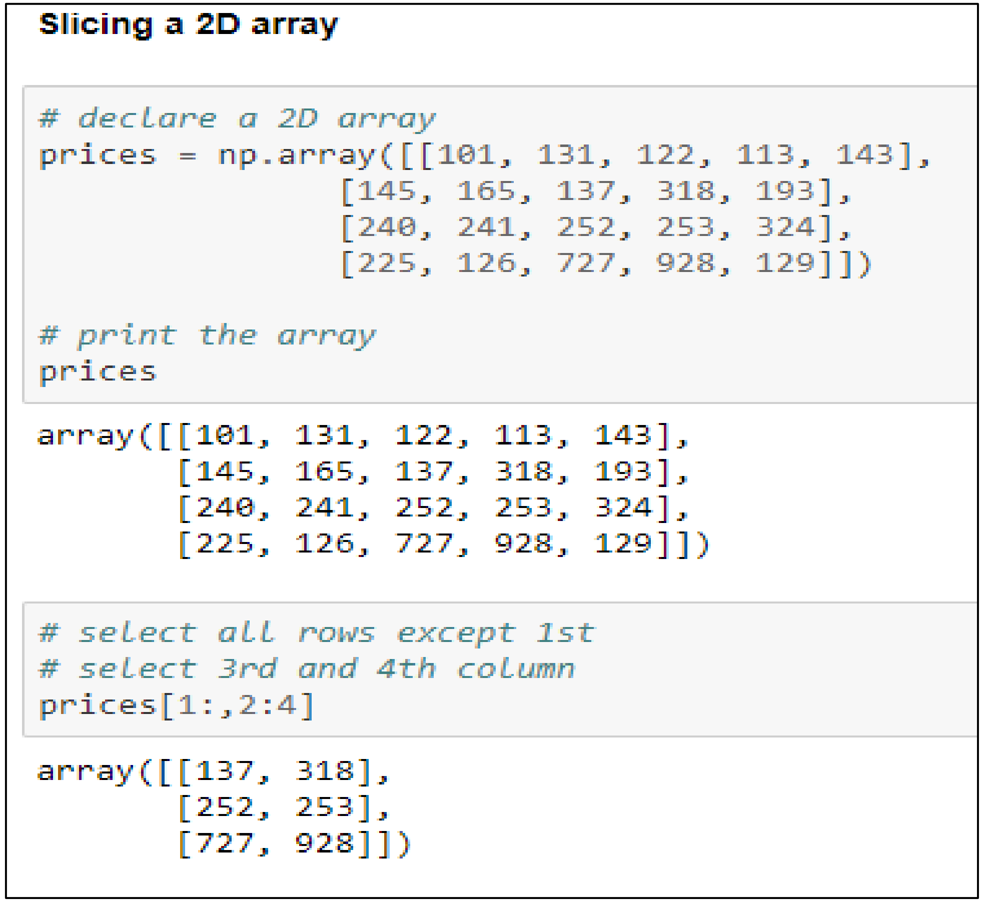
SLICING 2D ARRAY

- Slicing in 2d array returns a sub-matrix of the original matrix



```
# create a 2d array
age = np.array([(73, 68, 54), (21, 36, 19)])

# print the 2d array
print("Original array: ", age)

# retrieve the elements in the 1st row and first two columns
print("\nElements from first row and first two columns: ", age[0,0:2])

# retrieve the elements from 2nd row and 2nd and 3rd column
print("\nElements from second row and last two columns: ", age[1,1:])
```

```
Original array:  [[73 68 54]
 [21 36 19]]

Elements from first row and first two columns:  [73 68]

Elements from second row and last two columns:  [36 19]
```



Slicing a 2D array

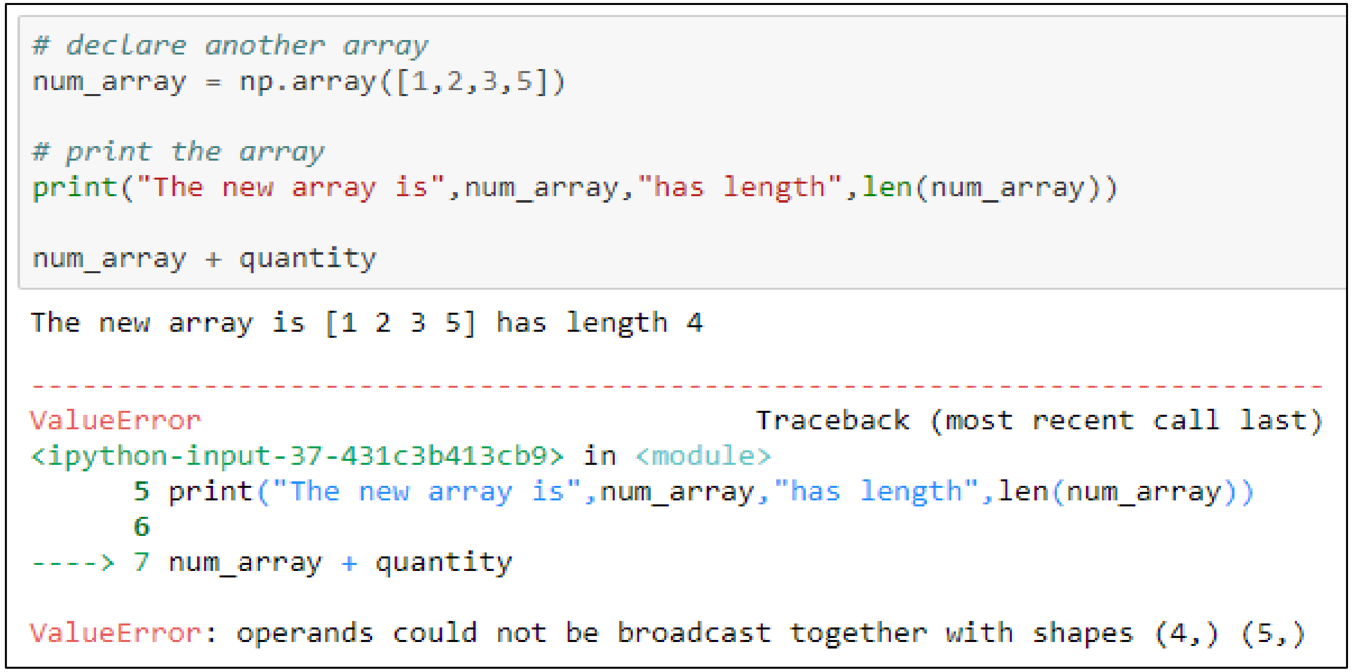
```
# declare a 2D array
prices = np.array([[101, 131, 122, 113, 143],
                   [145, 165, 137, 318, 193],
                   [240, 241, 252, 253, 324],
                   [225, 126, 727, 928, 129]])

# print the array
prices
```

```
array([[101, 131, 122, 113, 143],
       [145, 165, 137, 318, 193],
       [240, 241, 252, 253, 324],
       [225, 126, 727, 928, 129]])
```

```
# select all rows except 1st
# select 3rd and 4th column
prices[1:,2:4]
```

```
array([[137, 318],
       [252, 253],
       [727, 928]])
```

The index returns an element of the array, the slice returns a list of elements.



```
# declare another array
num_array = np.array([1,2,3,5])

# print the array
print("The new array is",num_array,"has length",len(num_array))

num_array + quantity
```

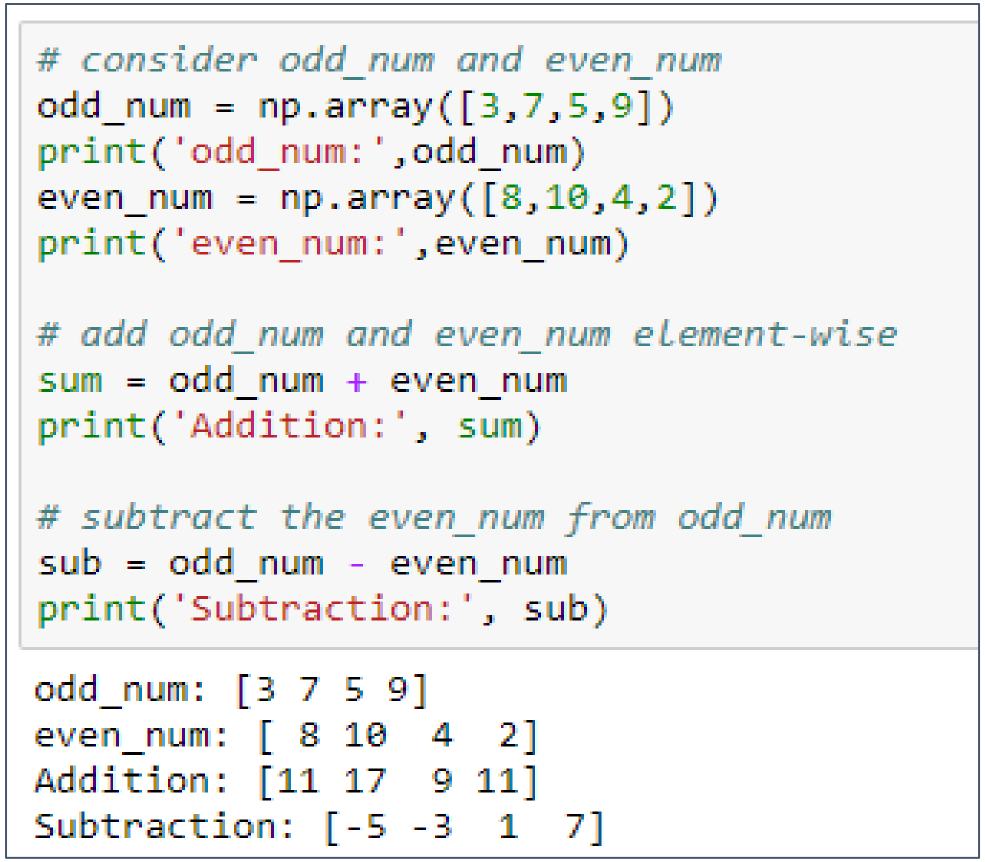
```
The new array is [1 2 3 5] has length 4

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-37-431c3b413cb9> in <module>
      5 print("The new array is",num_array,"has length",len(num_array))
      6 
----> 7 num_array + quantity

ValueError: operands could not be broadcast together with shapes (4,) (5,)
```

If you try to add arrays with different dimensions, you get an error.

Array Operations

ARITHMETIC OPERATIONS

Addition and Subtraction of 1D array

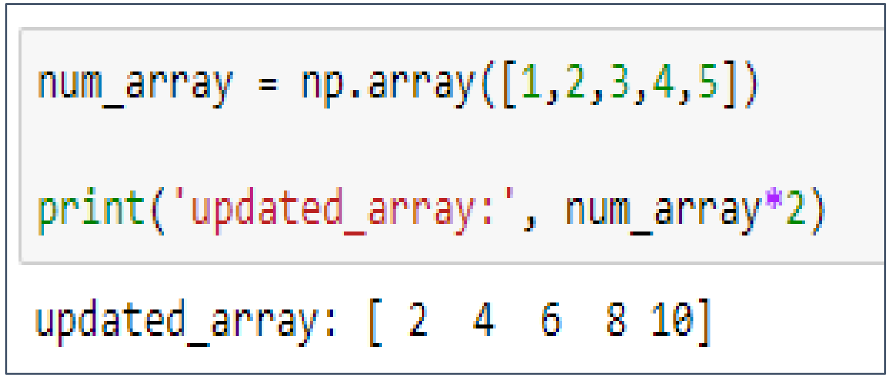


```
# consider odd_num and even_num
odd_num = np.array([3,7,5,9])
print('odd_num:',odd_num)
even_num = np.array([8,10,4,2])
print('even_num:',even_num)

# add odd_num and even_num element-wise
sum = odd_num + even_num
print('Addition:', sum)

# subtract the even_num from odd_num
sub = odd_num - even_num
print('Subtraction:', sub)
```

```
odd_num: [3 7 5 9]
even_num: [ 8 10  4  2]
Addition: [11 17  9 11]
Subtraction: [-5 -3  1  7]
```

ARITHMETIC OPERATIONS

Multiply each element in the array by 2



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

print('updated_array:', num_array*2)
```

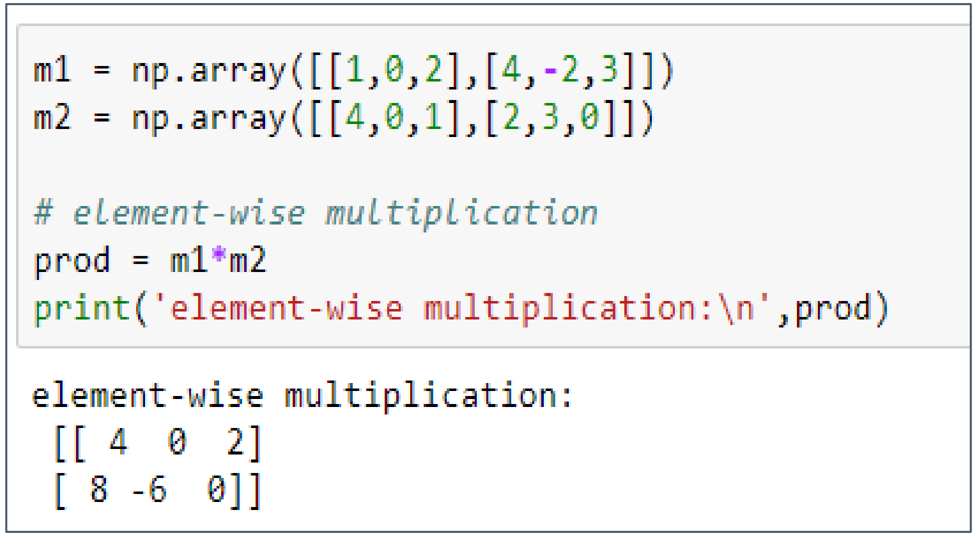
```
updated_array: [ 2  4  6  8 10]
```

ARITHMETIC OPERATIONS

Element-wise multiplication of two 3x3 matrices

$$\begin{bmatrix} 1 & 0 & 2 \\ 4 & -2 & 3 \end{bmatrix} \circ \begin{bmatrix} 4 & 0 & 1 \\ 2 & 3 & 0 \end{bmatrix} = \begin{bmatrix} 1X4 & 0X0 & 2X1 \\ 4X2 & -2X3 & 3X0 \end{bmatrix}$$

m1 m2 m1 * m2



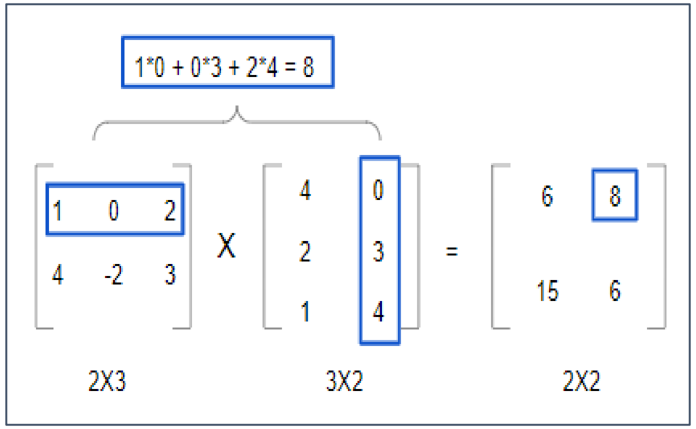
```
m1 = np.array([[1,0,2],[4,-2,3]])
m2 = np.array([[4,0,1],[2,3,0]])

# element-wise multiplication
prod = m1*m2
print('element-wise multiplication:\n',prod)
```

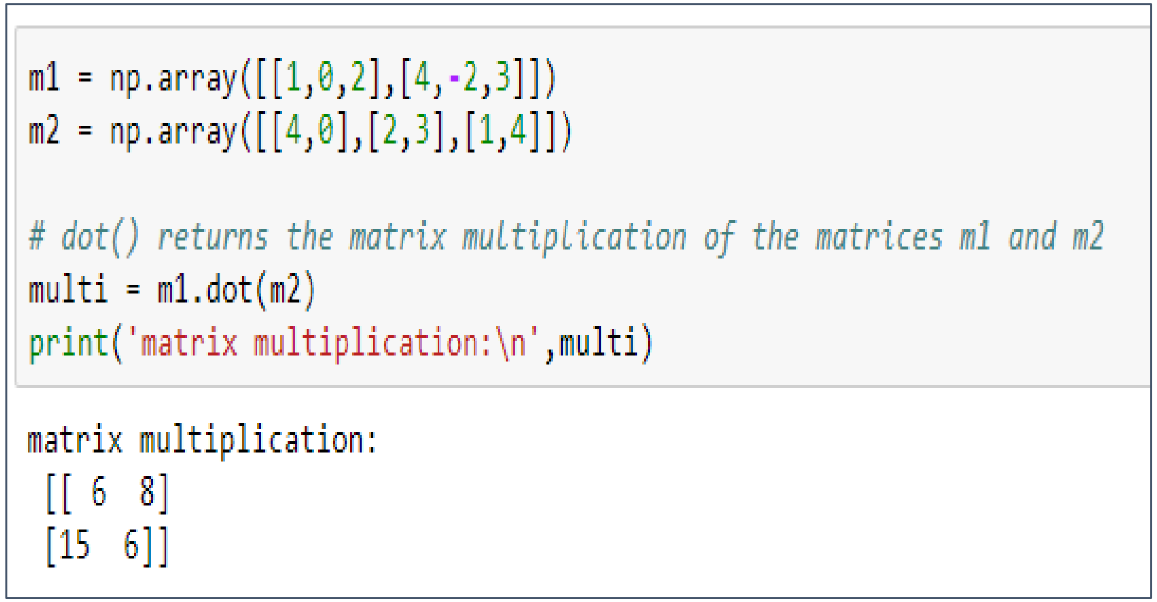
```
element-wise multiplication:
 [[ 4  0  2]
 [ 8 -6  0]]
```

This product is also known as ‘Hadamard Product’

ARITHMETIC OPERATIONS

Matrix multiplication of two 3x3 matrices



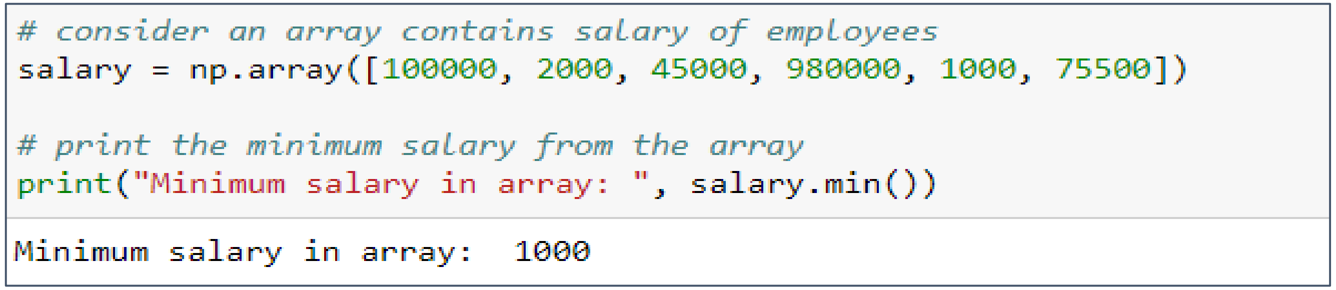


```
m1 = np.array([[1,0,2],[4,-2,3]])
m2 = np.array([[4,0],[2,3],[1,4]])

# dot() returns the matrix multiplication of the matrices m1 and m2
multi = m1.dot(m2)
print('matrix multiplication:\n',multi)
```

```
matrix multiplication:
 [[ 6  8]
 [15  6]]
```

ARITHMETIC OPERATIONS

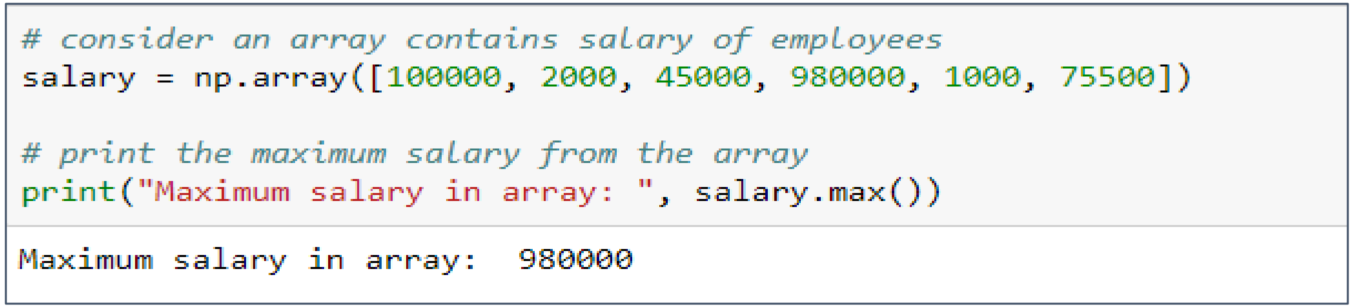
- The min() returns the minimum value present in the array



```
# consider an array contains salary of employees
salary = np.array([100000, 2000, 45000, 980000, 1000, 75500])

# print the minimum salary from the array
print("Minimum salary in array: ", salary.min())
```

```
Minimum salary in array:  1000
```

- The max() returns the maximum value present in the array

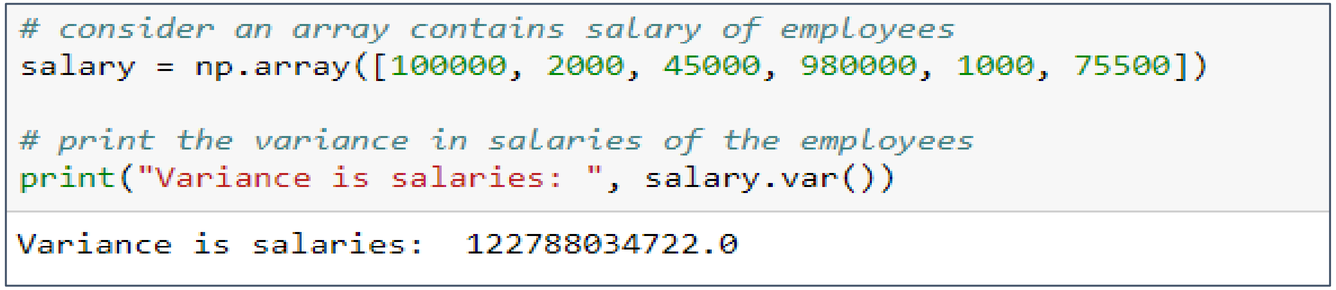


```
# consider an array contains salary of employees
salary = np.array([100000, 2000, 45000, 980000, 1000, 75500])

# print the maximum salary from the array
print("Maximum salary in array: ", salary.max())
```

```
Maximum salary in array:  980000
```

ARITHMETIC OPERATIONS

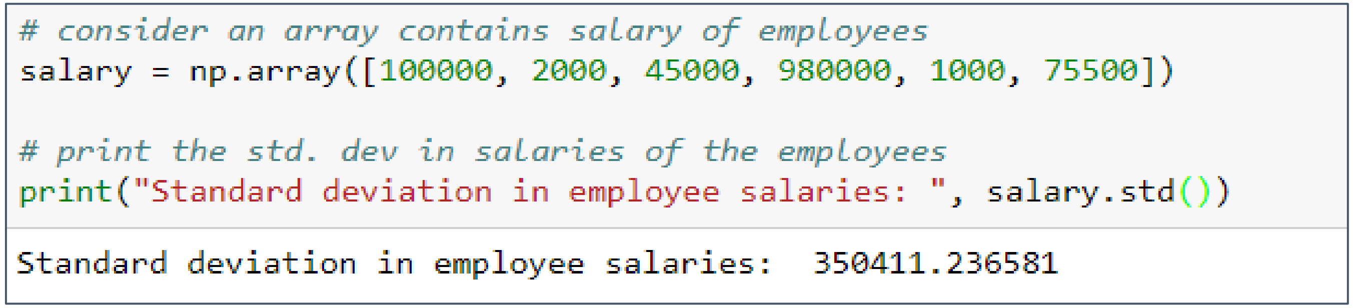
- The var() returns the variance of all the elements in the array



```
# consider an array contains salary of employees
salary = np.array([100000, 2000, 45000, 980000, 1000, 75500])

# print the variance in salaries of the employees
print("Variance is salaries: ", salary.var())
```

```
Variance is salaries:  122788034722.0
```

- The std() returns the standard deviation of all the elements in the array

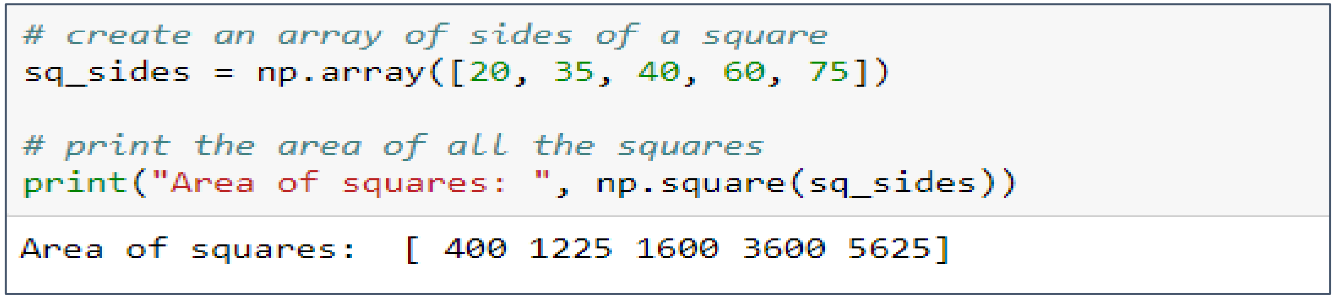


```
# consider an array contains salary of employees
salary = np.array([100000, 2000, 45000, 980000, 1000, 75500])

# print the std. dev in salaries of the employees
print("Standard deviation in employee salaries: ", salary.std())
```

```
Standard deviation in employee salaries:  350411.236581
```

ARITHMETIC OPERATIONS

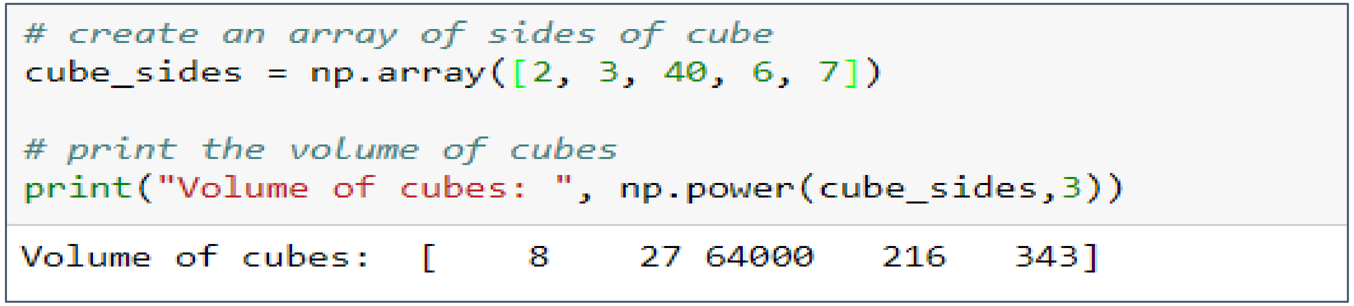
- The np.square() returns the square of the elements



```
# create an array of sides of a square
sq_sides = np.array([20, 35, 40, 60, 75])

# print the area of all the squares
print("Area of squares: ", np.square(sq_sides))
```

```
Area of squares:  [ 400 1225 1600 3600 5625]
```

- The np.power() is used to raise the numbers in the array to the given value

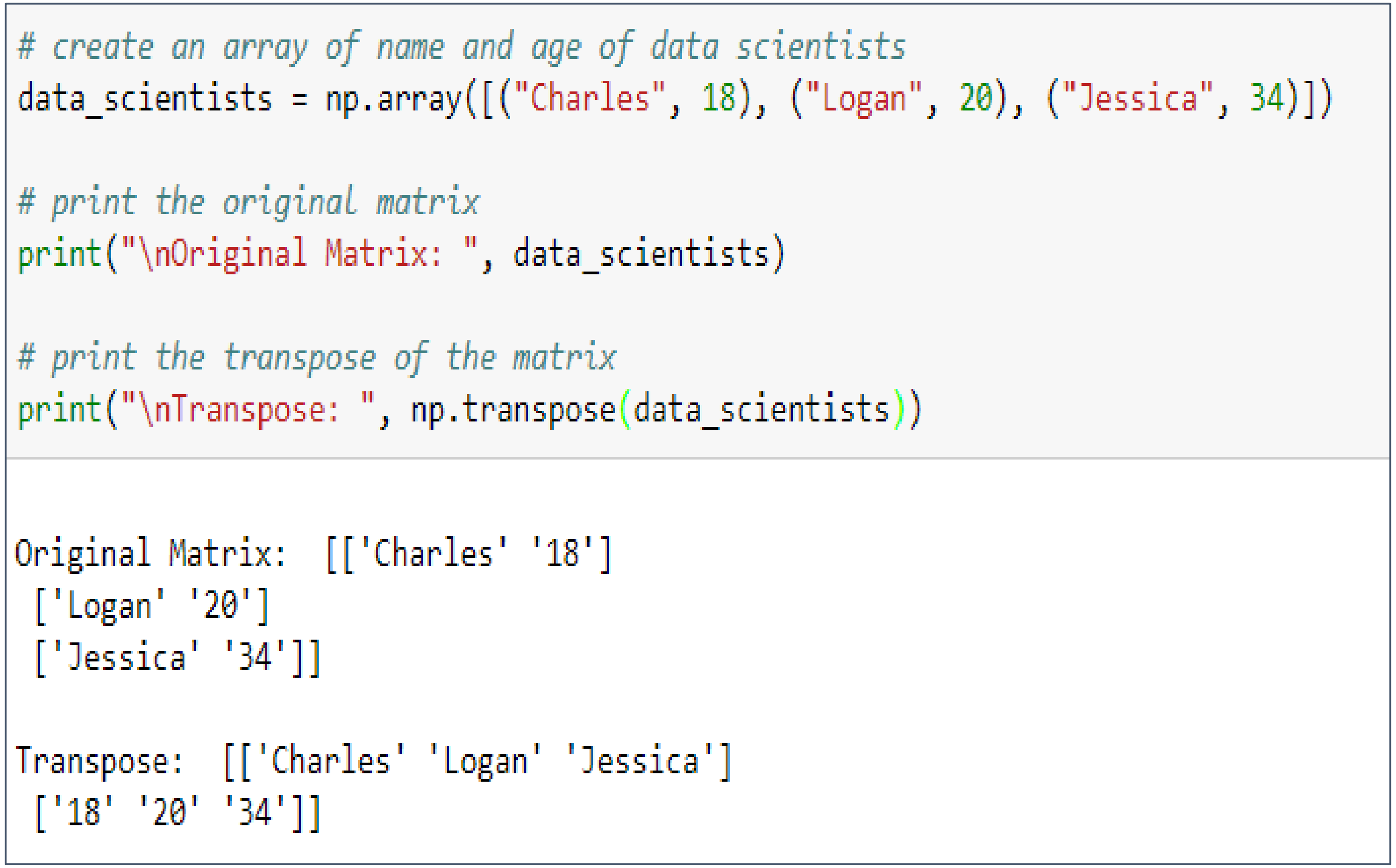


```
# create an array of sides of cube
cube_sides = np.array([2, 3, 40, 6, 7])

# print the volume of cubes
print("Volume of cubes: ", np.power(cube_sides,3))
```

```
Volume of cubes:  [    8    27 64000   216   343]
```

ARITHMETIC OPERATIONS

- The np.transpose() reverses the dimension of the array

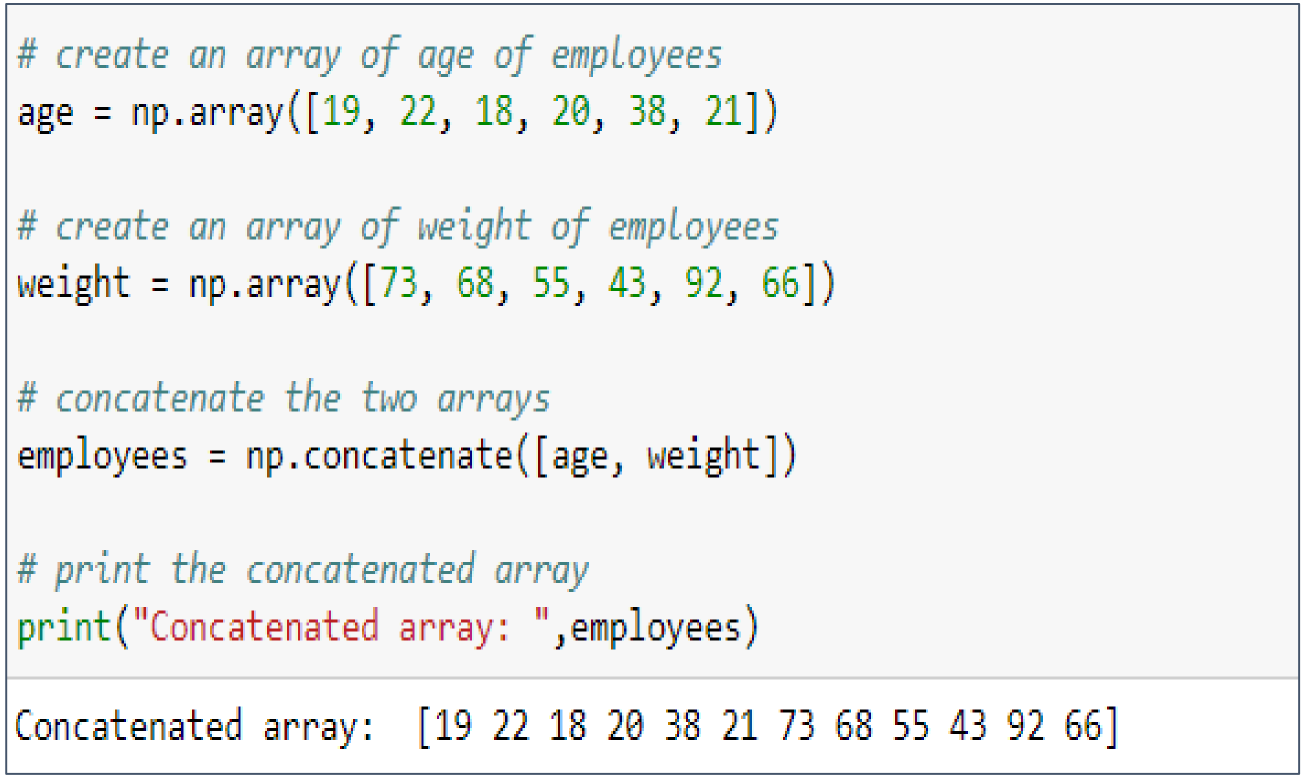


```
# create an array of name and age of data scientists
data_scientists = np.array([("Charles", 18), ("Logan", 20), ("Jessica", 34)])

# print the original matrix
print("\nOriginal Matrix: ", data_scientists)

# print the transpose of the matrix
print("\nTranspose: ", np.transpose(data_scientists))
```

```

Original Matrix:  [['Charles' '18']
 ['Logan' '20']
 ['Jessica' '34']]

Transpose:  [['Charles' 'Logan' 'Jessica']
 ['18' '20' '34']]
```

CONCATENATE 1D ARRAY

- Two or more arrays will get joined along existing (first) axis, provided they have the same shape

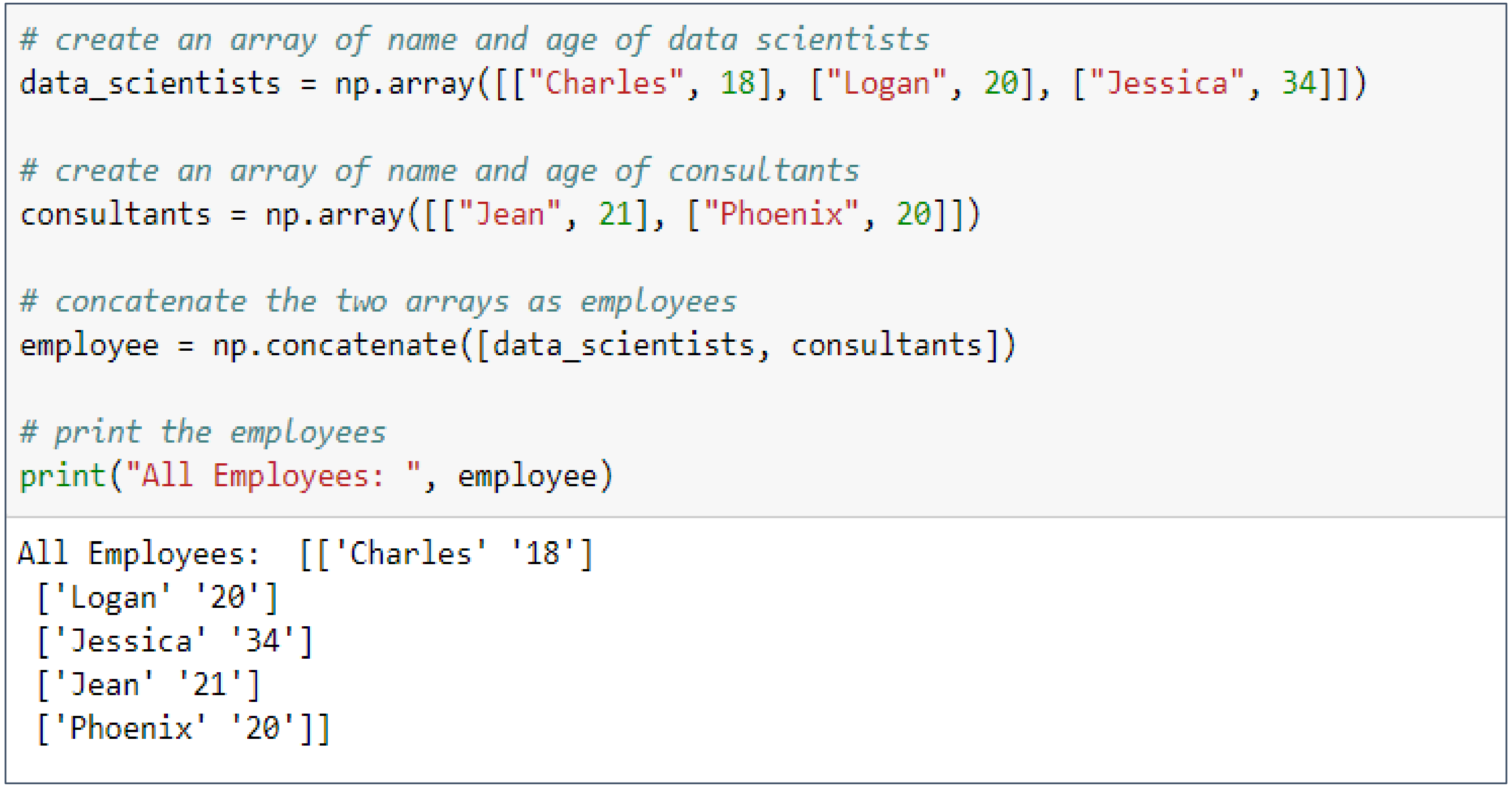


```
# create an array of age of employees
age = np.array([19, 22, 18, 20, 38, 21])

# create an array of weight of employees
weight = np.array([73, 68, 55, 43, 92, 66])

# concatenate the two arrays
employees = np.concatenate([age, weight])

# print the concatenated array
print("Concatenated array: ",employees)
```

```
Concatenated array:  [19 22 18 20 38 21 73 68 55 43 92 66]
```

CONCATENATE 2D ARRAY

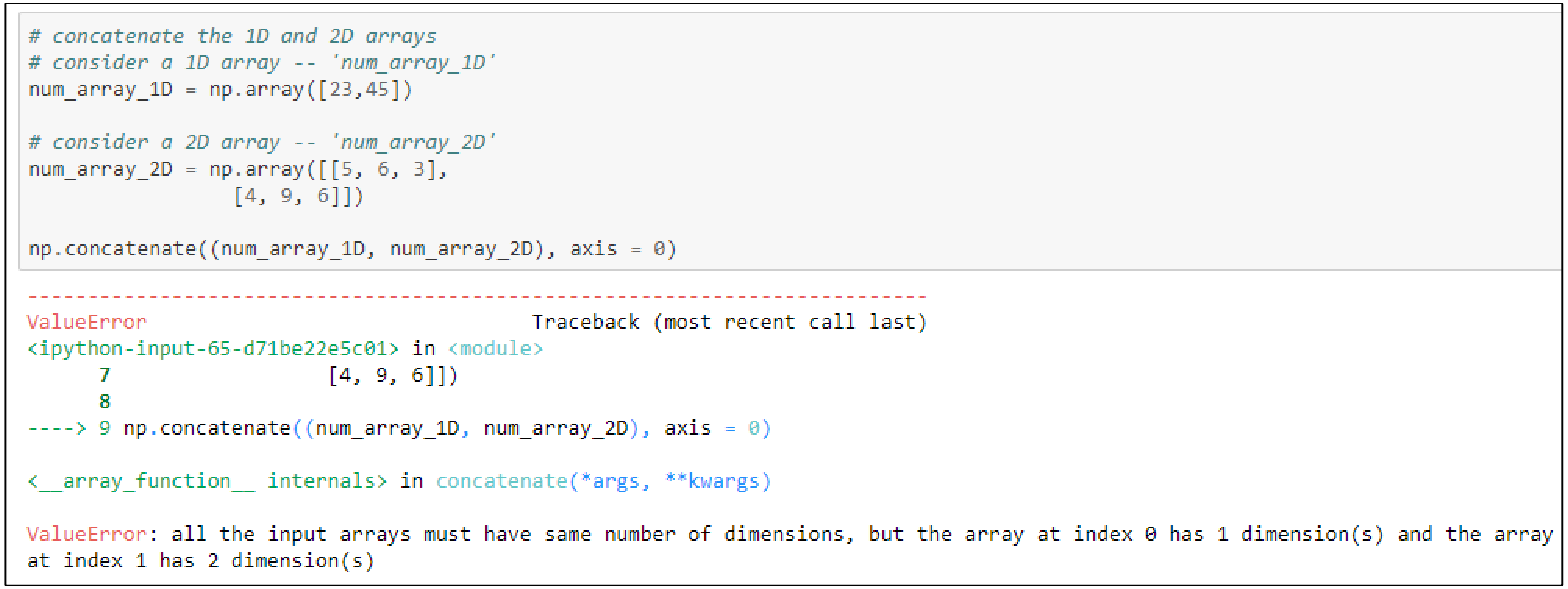
- We can concatenate 2D arrays either along rows (axis = 0) or columns (axis = 1), provided they have same shape



```
# create an array of name and age of data scientists
data_scientists = np.array([["Charles", 18], ["Logan", 20], ["Jessica", 34]])

# create an array of name and age of consultants
consultants = np.array([["Jean", 21], ["Phoenix", 20]])

# concatenate the two arrays as employees
employee = np.concatenate([data_scientists, consultants])

# print the employees
print("All Employees: ", employee)
```

```
All Employees:  [['Charles' '18']
 ['Logan' '20']
 ['Jessica' '34']
 ['Jean' '21']
 ['Phoenix' '20']]
```

We can not concatenate the arrays with different dimensions

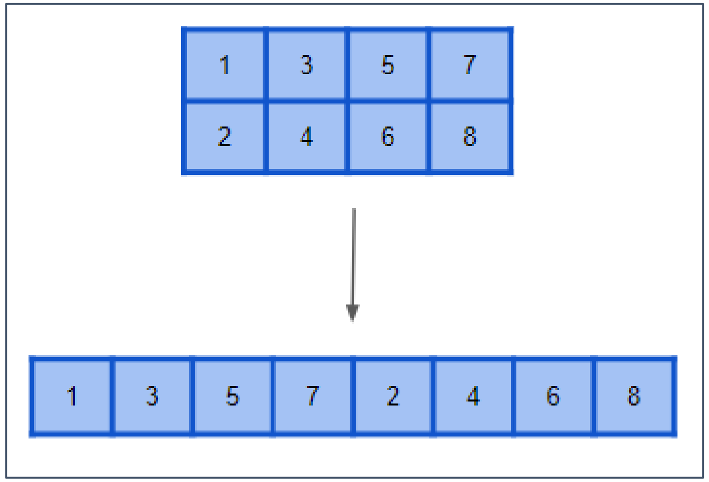


```
# concatenate the 1D and 2D arrays
# consider a 1D array -- 'num_array_1D'
num_array_1D = np.array([23,45])

# consider a 2D array -- 'num_array_2D'
num_array_2D = np.array([[5, 6, 3],
                         [4, 9, 6]])

np.concatenate((num_array_1D, num_array_2D), axis = 0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-65-d71be22e5c01> in <module>
      7                          [4, 9, 6]])
      8 
----> 9 np.concatenate((num_array_1D, num_array_2D), axis = 0)

<__array_function__ internals> in concatenate(*args, **kwargs)

ValueError: all the input arrays must have same number of dimensions, but the array at index 0 has 1 dimension(s) and the array
at index 1 has 2 dimension(s)
```

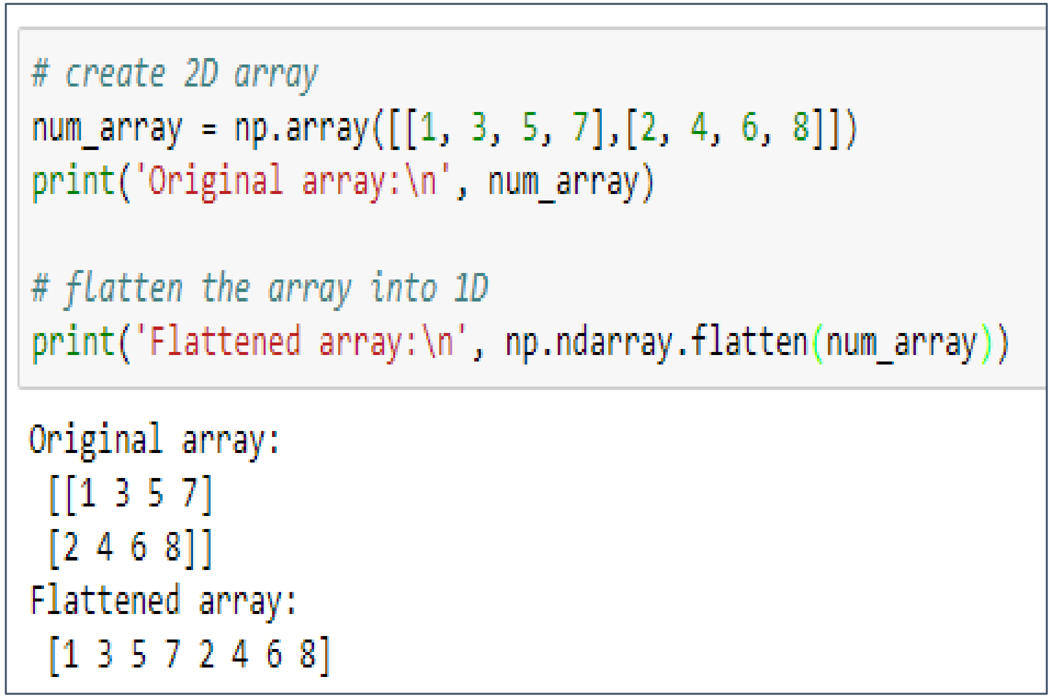
FLATTEN THE ARRAY

- The flatten() function collapses the original array into a single dimension



1	3	5	7
2	4	6	8

1	3	5	7	2	4	6	8

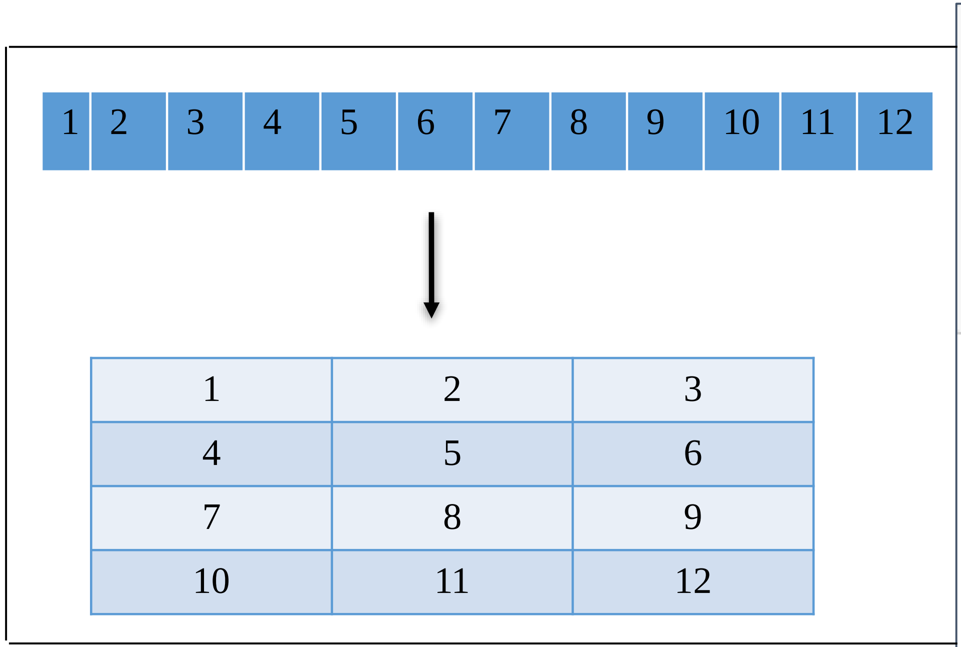


```
# create 2D array
num_array = np.array([[1, 3, 5, 7],[2, 4, 6, 8]])
print('Original array:\n', num_array)

# flatten the array into 1D
print('Flattened array:\n', np.ndarray.flatten(num_array))
```

```
Original array:
 [[1 3 5 7]
 [2 4 6 8]]
Flattened array:
 [1 3 5 7 2 4 6 8]
```

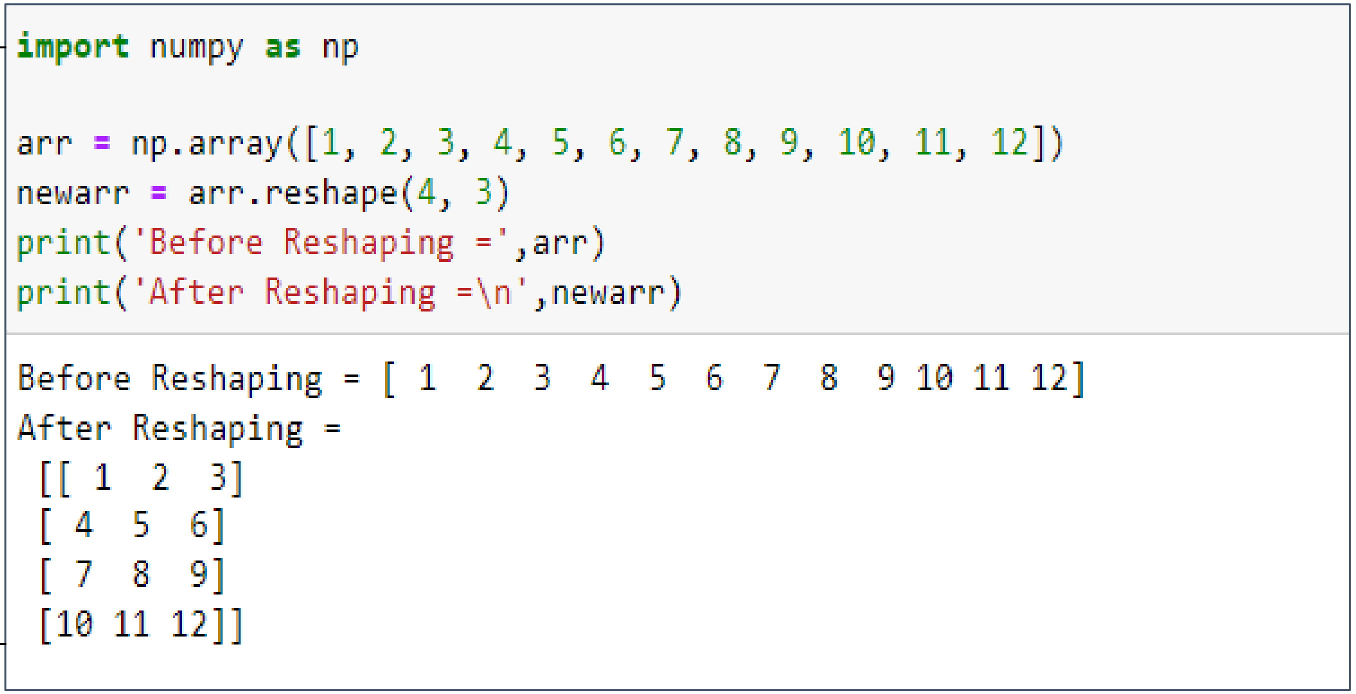
Reshape the Array

- Reshaping means changing the shape of an array.
- The shape of an array is the number of elements in each dimension.
- By reshaping we can add or remove dimensions or change the number of elements in each dimension.



1	2	3	4	5	6	7	8	9	10	11	12

1	2	3
4	5	6
7	8	9
10	11	12

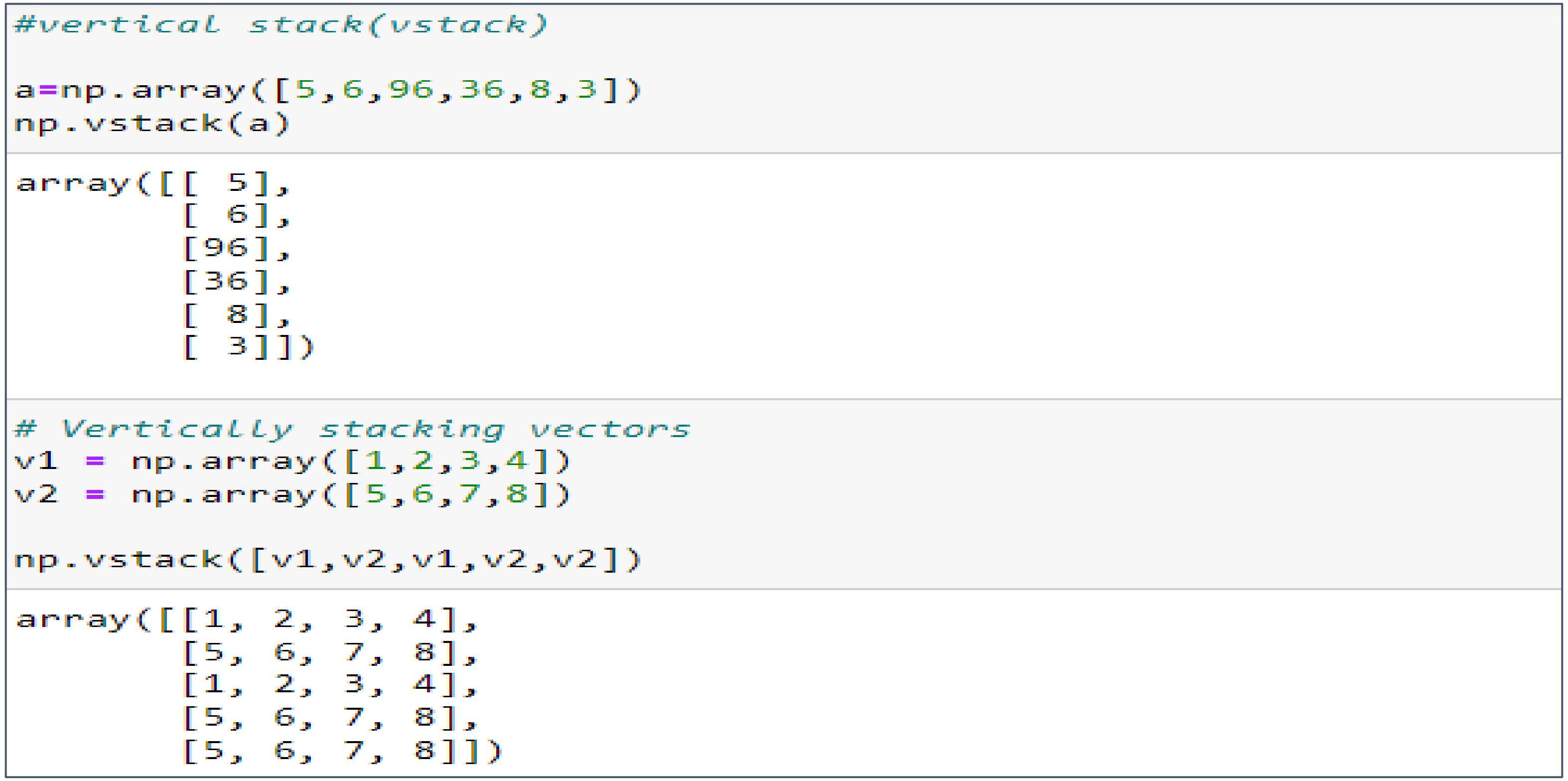


```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
newarr = arr.reshape(4, 3)
print('Before Reshaping =',arr)
print('After Reshaping =\n',newarr)
```

```
Before Reshaping = [ 1  2  3  4  5  6  7  8  9 10 11 12]
After Reshaping =
 [[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Vertical Stack (vstack)

- You can do vertical staking for one vector (vstack)
- If you want to perform for more than one vector you want to mention in the list
- If you are vertically stacking more than one array the size of array should be same



```
#vertical stack(vstack)

a=np.array([5,6,96,36,8,3])
np.vstack(a)
```

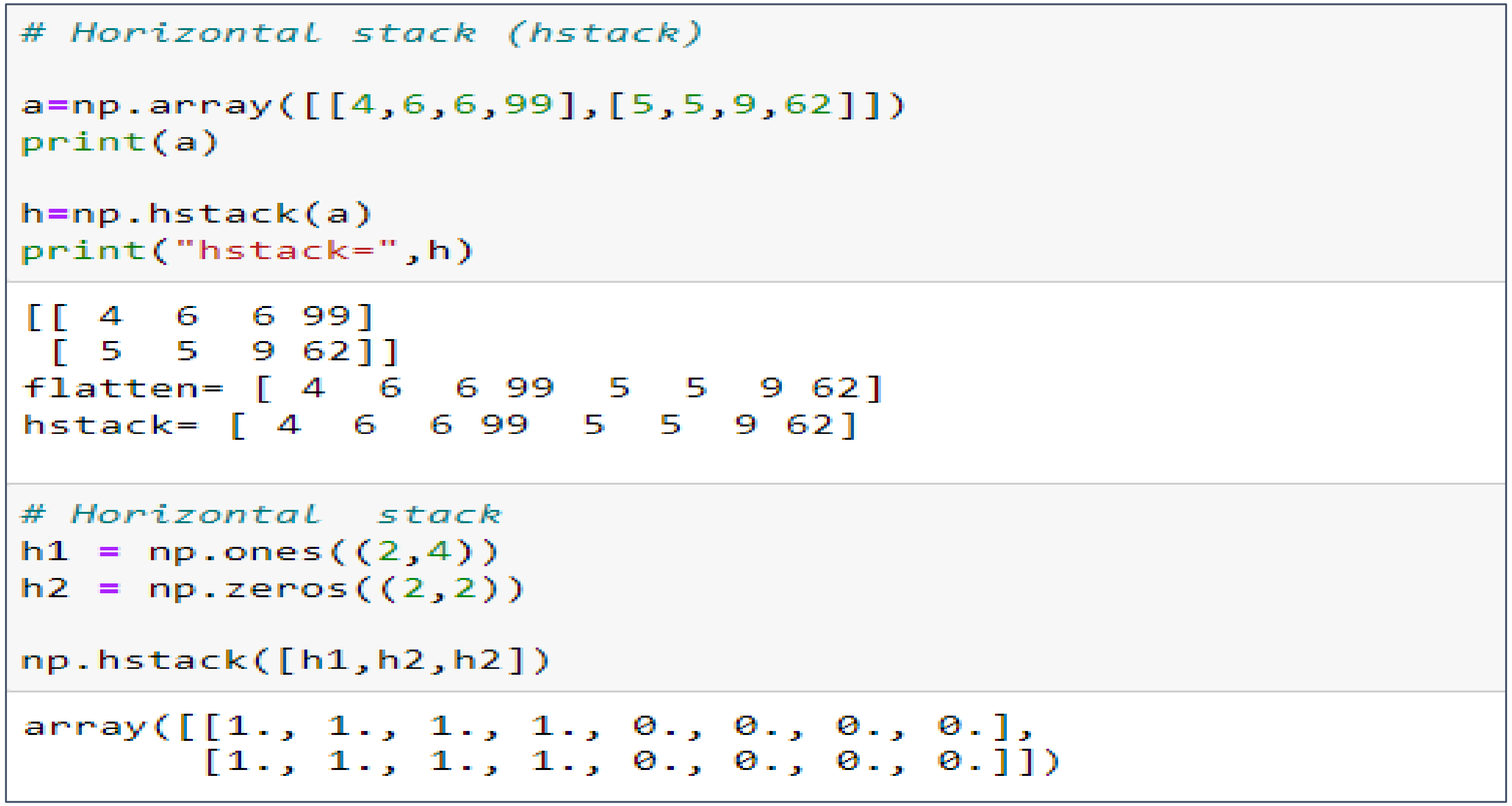
```
array([[ 5],
       [ 6],
       [96],
       [36],
       [ 8],
       [ 3]])
```

```
# Vertically stacking vectors
v1 = np.array([1,2,3,4])
v2 = np.array([5,6,7,8])

np.vstack([v1,v2,v1,v2,v2])
```

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

Horizontal Stack (hstack)

- You can do horizontal staking for one vector (hstack)
- If you want to perform for more than one vector you want to mention in the list
- If you are horizontally stacking more than one array the size of array may be different not an issue



```
# Horizontal stack (hstack)

a=np.array([[4,6,6,99],[5,5,9,62]])
print(a)

h=np.hstack(a)
print("hstack=",h)
```

```
[[ 4  6  6 99]
 [ 5  5  9 62]]
flatten= [ 4  6  6 99  5  5  9 62]
hstack= [ 4  6  6 99  5  5  9 62]
```

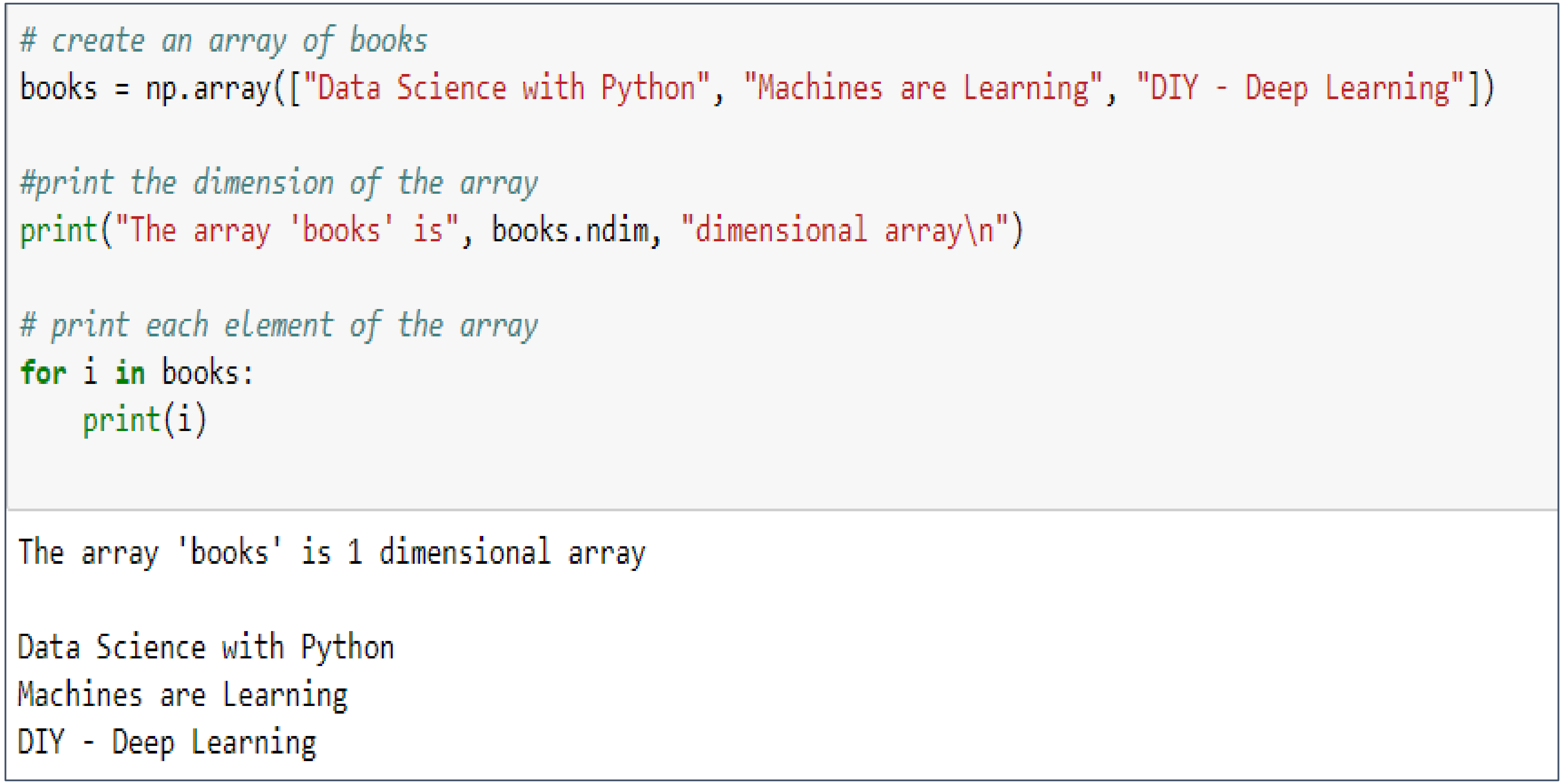
```
# Horizontal  stack
h1 = np.ones((2,4))
h2 = np.zeros((2,2))

np.hstack([h1,h2,h2])
```

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

Iterating through Arrays

ITERATING THROUGH 1D ARRAY

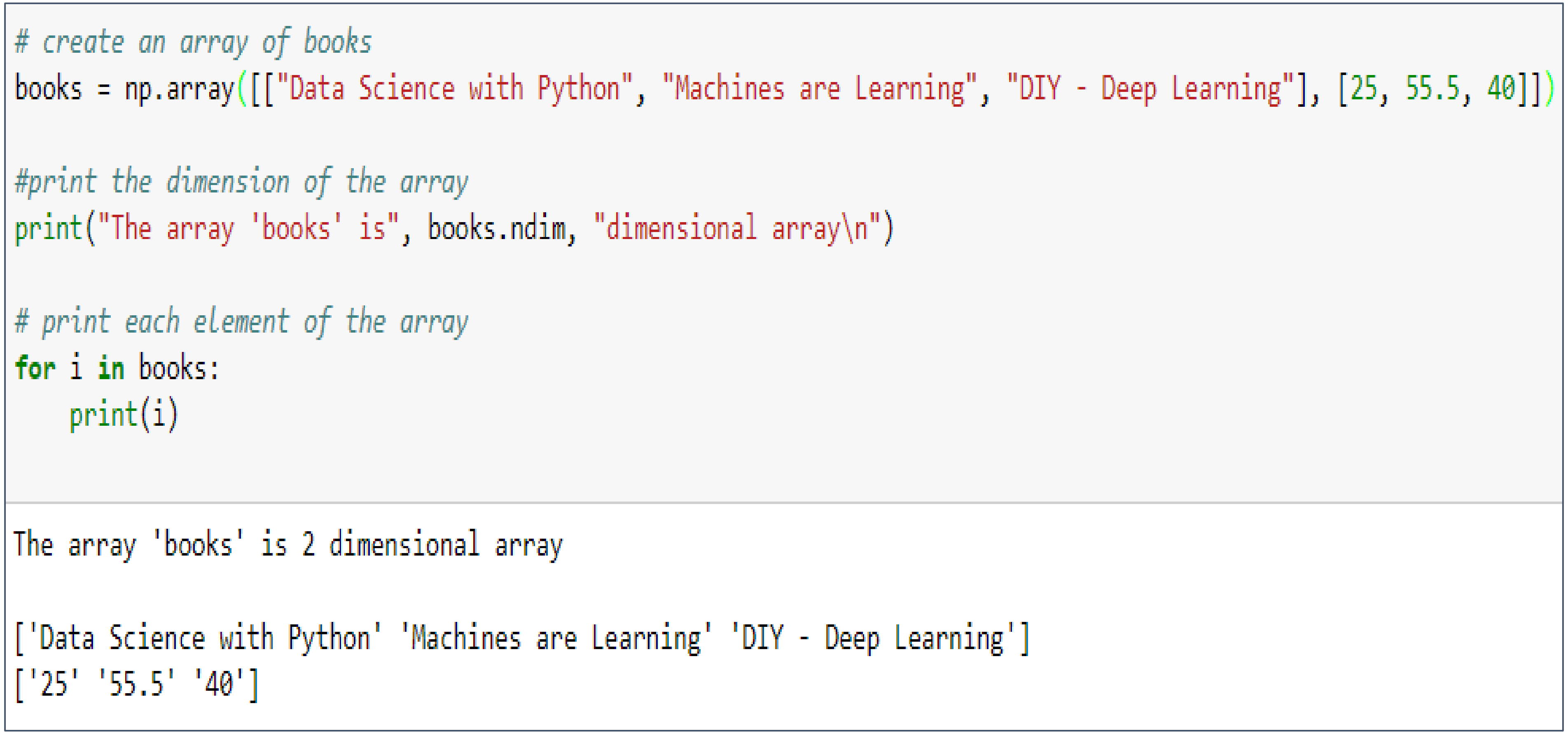
- The for loop can be used to iterate through the array elements



```
# create an array of books
books = np.array(["Data Science with Python", "Machines are Learning", "DIY - Deep Learning"])

#print the dimension of the array
print("The array 'books' is", books.ndim, "dimensional array\n")

# print each element of the array
for i in books:
    print(i)
```

```
The array 'books' is 1 dimensional array

Data Science with Python
Machines are Learning
DIY - Deep Learning
```

ITERATING THROUGH 2D ARRAY

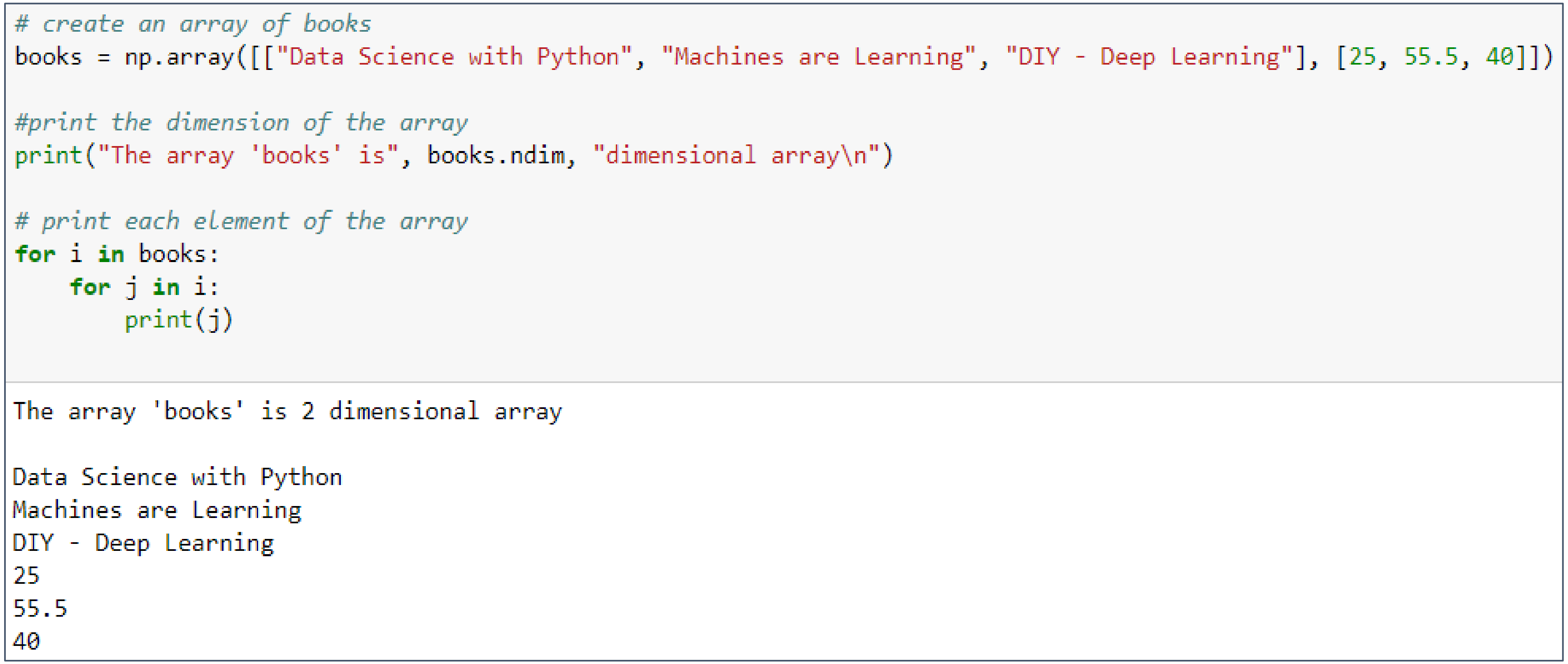


```
# create an array of books
books = np.array([["Data Science with Python", "Machines are Learning", "DIY - Deep Learning"], [25, 55.5, 40]])

#print the dimension of the array
print("The array 'books' is", books.ndim, "dimensional array\n")

# print each element of the array
for i in books:
    print(i)
```

```
The array 'books' is 2 dimensional array

['Data Science with Python' 'Machines are Learning' 'DIY - Deep Learning']
['25' '55.5' '40']
```

ITERATING THROUGH 2D ARRAY USING NESTED FOR LOOP

- To print each element in the 2D array, use nested for loop

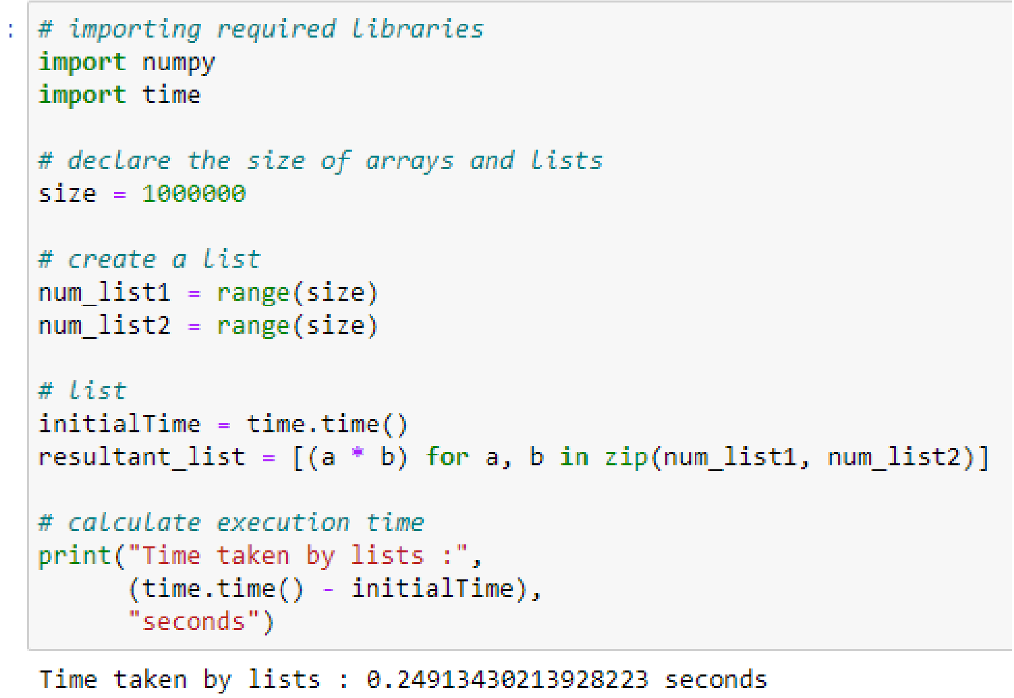


```python
# create an array of books
books = np.array([["Data Science with Python", "Machines are Learning", "DIY - Deep Learning"], [25, 55.5, 40]])

#print the dimension of the array
print("The array 'books' is", books.ndim, "dimensional array\n")

# print each element of the array
for i in books:
    for j in i:
        print(j)
```

```
The array 'books' is 2 dimensional array

Data Science with Python
Machines are Learning
DIY - Deep Learning
25
55.5
40
```

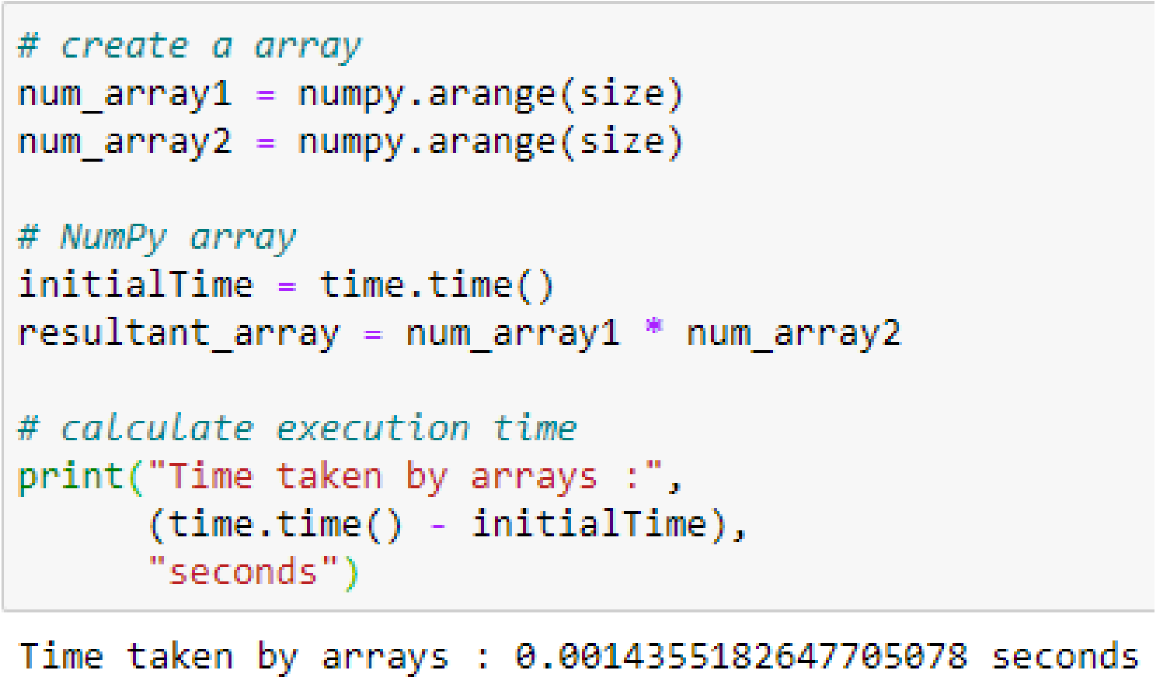
DID YOU KNOW?

NumPy arrays are faster than list. The below code shows that NumPy arrays are very much faster than the lists in python.



```
# importing required libraries
import numpy
import time

# declare the size of arrays and lists
size = 1000000

# create a list
num_list1 = range(size)
num_list2 = range(size)

# list
initialTime = time.time()
resultant_list = [(a * b) for a, b in zip(num_list1, num_list2)]

# calculate execution time
print("Time taken by lists :",
      (time.time() - initialTime),
      "seconds")
```

```
Time taken by lists : 0.24913430213928223 seconds
```



```
# create a array
num_array1 = numpy.arange(size)
num_array2 = numpy.arange(size)

# NumPy array
initialTime = time.time()
resultant_array = num_array1 * num_array2

# calculate execution time
print("Time taken by arrays :",
      (time.time() - initialTime),
      "seconds")
```

```
Time taken by arrays : 0.0014355182647705078 seconds
```