Leveling test

The content of each lecture :

1. 20min:Quiz on last lecture
2. Lecture
3. Lab: to be delivered by the end of lab
4. Presentation : to be delivered at certain deadline

NumPy Basics

Introduction to NumPy

NumPy is the fundamental package for scientific computing in Python. NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data. Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences. NumPy is not another programming language but a Python extension module. It provides fast and efficient operations on arrays of homogeneous data.

Some important points about Numpy arrays:

•We can create a N-dimensional array in python using numpy.array().

•Array are by default Homogeneous, which means data inside an array must be of the same Datatype. (Note you can also create a structured array in python).

•Element wise operation is possible.

•Numpy array has the various function, methods, and variables, to ease our task of matrix computation.

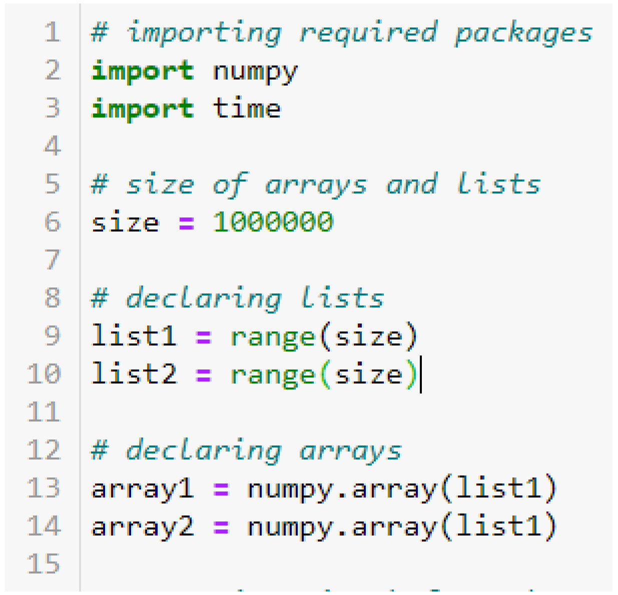
•Elements of an array are stored contiguously in memory.
For example, all rows of a two dimensioned array must have the same number of columns. Or a three dimensioned array must have the same number of rows and columns on each card.

Session Objectives

At the end of this session, you will be able:

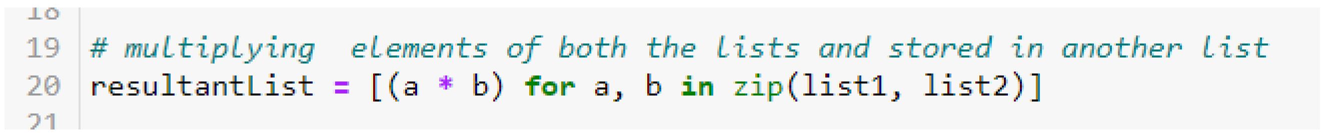
- ❑ Explain what for we use NumPy: the advantages of NumPy array over pure python data
- ❑ Create a 1D-Array and 2D-Array from common NumPy functions
- ❑ Manipulate different the standard data types of NumPy
- ❑ understand the most basics attribute of array
- ❑ Understand the most basics universal functions of NumPy
- ❑ Understand how to Transform arrays: Boolean indexing of arrays, slice arrays, reshape arrays

❑ The advantages of NumPy array over pure python data



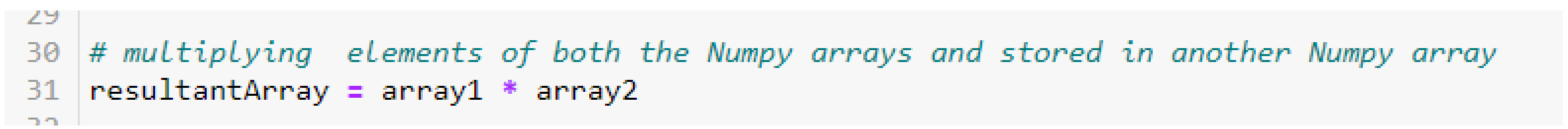
```python
# importing required packages
import numpy
import time

# size of arrays and lists
size = 1000000

# declaring lists
list1 = range(size)
list2 = range(size)

# declaring arrays
array1 = numpy.array(list1)
array2 = numpy.array(list1)

```

Element wise multiplication >> List



```python
# multiplying  elements of both the lists and stored in another list
resultantList = [(a * b) for a, b in zip(list1, list2)]
```

Element wise multiplication >> NumPy array



```python
# multiplying  elements of both the Numpy arrays and stored in another Numpy array
resultantArray = array1 * array2
```

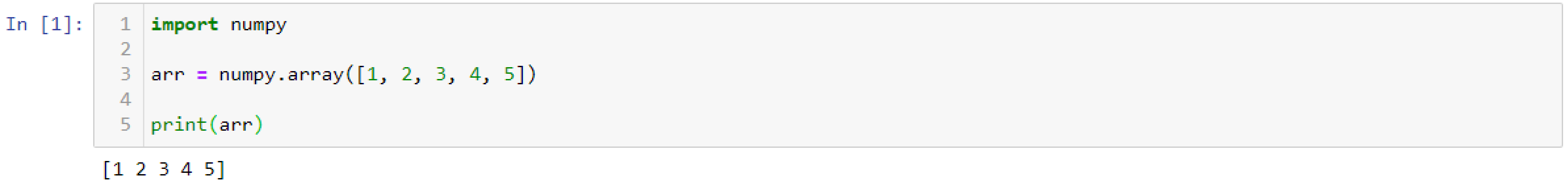
Numpy data structures perform better in:

1. Size - Numpy data structures take up less space
2. Performance - they have a need for speed and are faster than lists
3. Functionality - SciPy and NumPy have optimized functions such as linear algebra operations built in.

❑ Numpy(NumPy package , how to use it , Checking NumPy Version)

- NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.
- The source code for NumPy is located at this github repository https://github.com/numpy/numpy

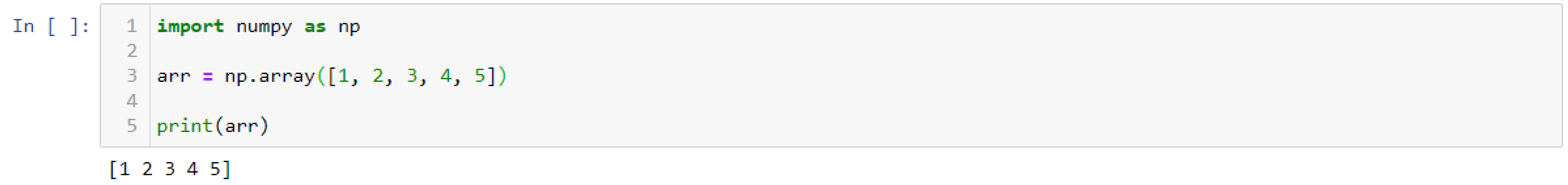
❖ Example



```
In [1]: import numpy

        arr = numpy.array([1, 2, 3, 4, 5])

        print(arr)

[1 2 3 4 5]
```

NumPy as np

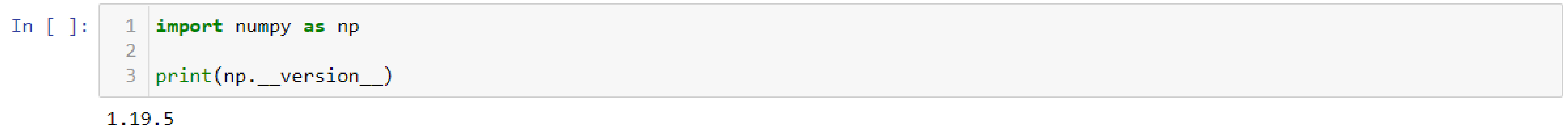
NumPy is usually imported under the np alias.



```
In [ ]:
import numpy as np

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

print(arr)
```

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

Checking NumPy Version



```
In [ ]:
import numpy as np

print(np.__version__)
```

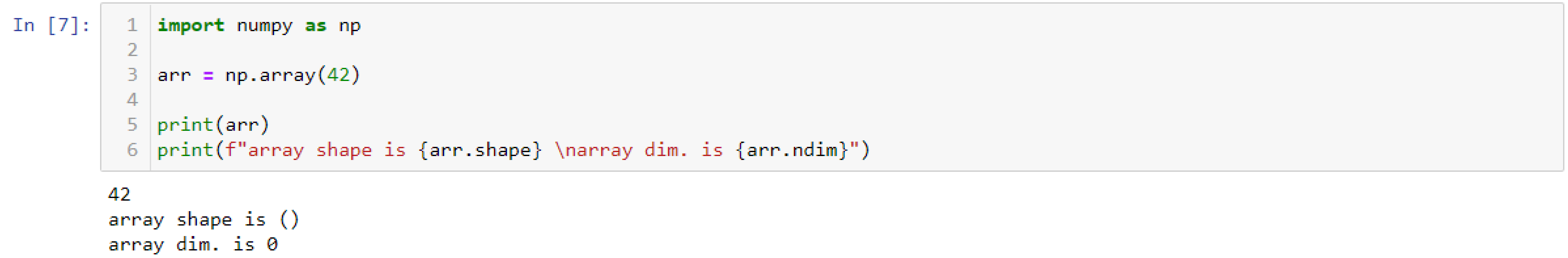
```
1.19.5
```

Create a N-dimension array from common NumPy functions

A dimension in arrays is one level of array depth (nested arrays)

1. 0-D Arrays

0-D arrays, or Scalars, are the elements in an array

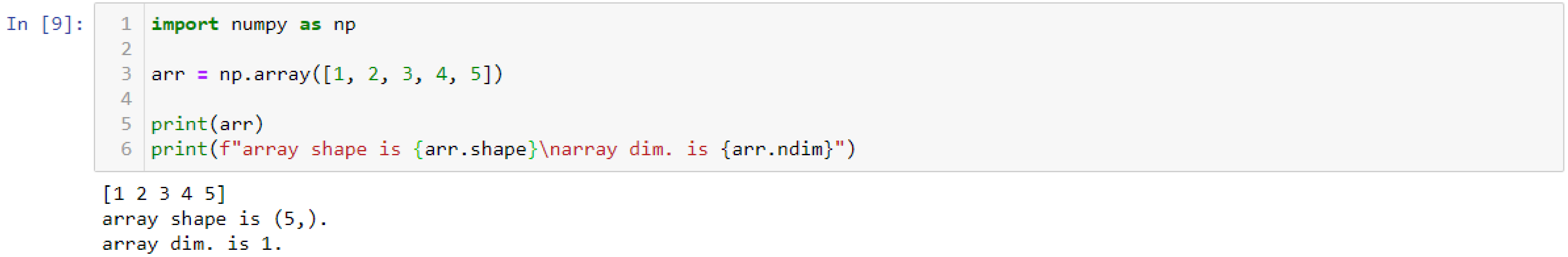


```
In [7]:
import numpy as np

arr = np.array(42)

print(arr)
print(f"array shape is {arr.shape} \narray dim. is {arr.ndim}")
```

```
42
array shape is ()
array dim. is 0
```

2. 1-D Arrays

- An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
- These are the most common and basic arrays.

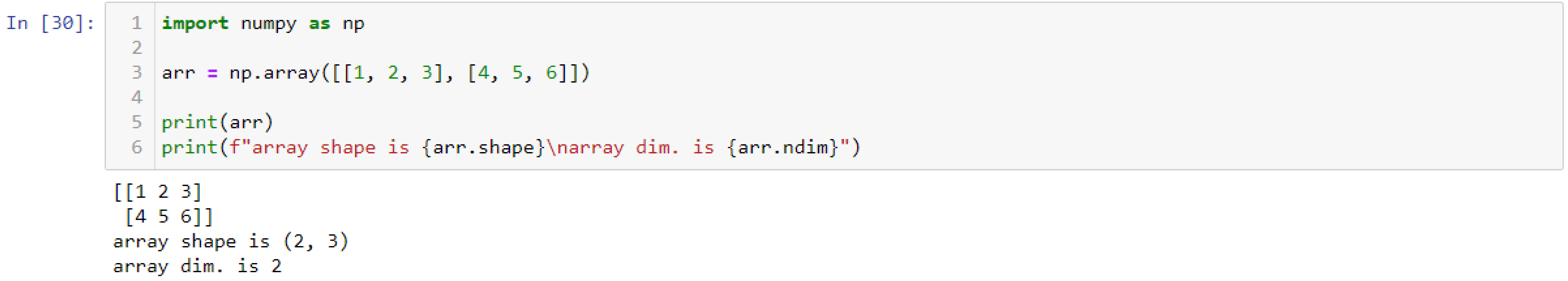


```
In [9]:
import numpy as np

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

print(arr)
print(f"array shape is {arr.shape}\narray dim. is {arr.ndim}")
```

```
[1 2 3 4 5]
array shape is (5,).
array dim. is 1.
```

3. 2-D arrays

- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.



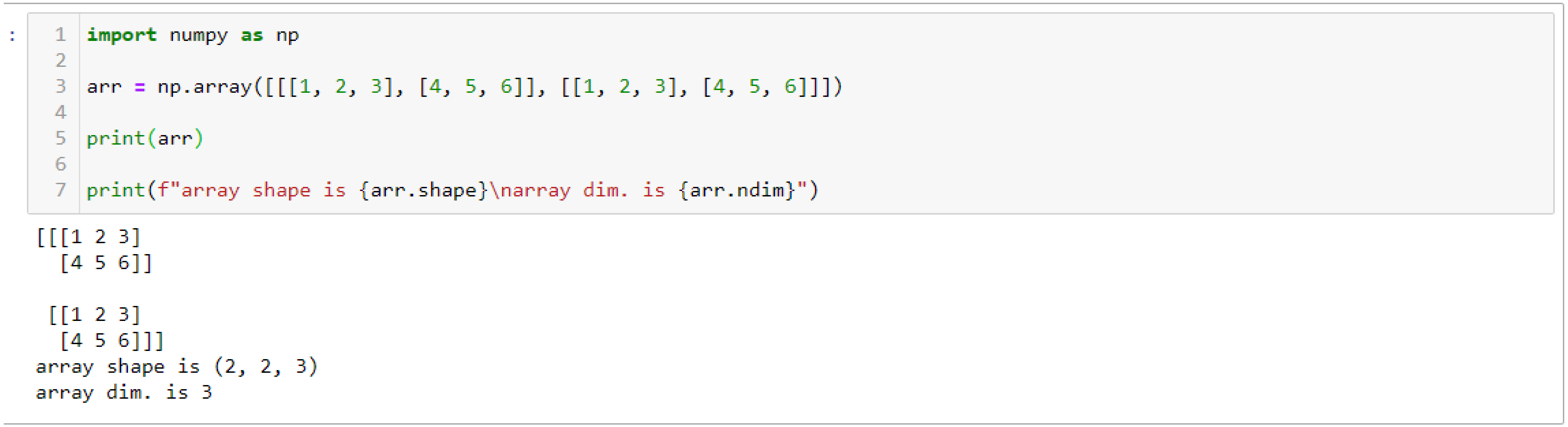
In [30]:

```python
import numpy as np

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

print(arr)
print(f"array shape is {arr.shape}\narray dim. is {arr.ndim}")
```

```
[[1 2 3]
 [4 5 6]]
array shape is (2, 3)
array dim. is 2
```

4. 3-D arrays

- An array that has 2-D arrays (matrices) as its elements is called 3-D array.
- These are often used to represent a 3rd order tensor.

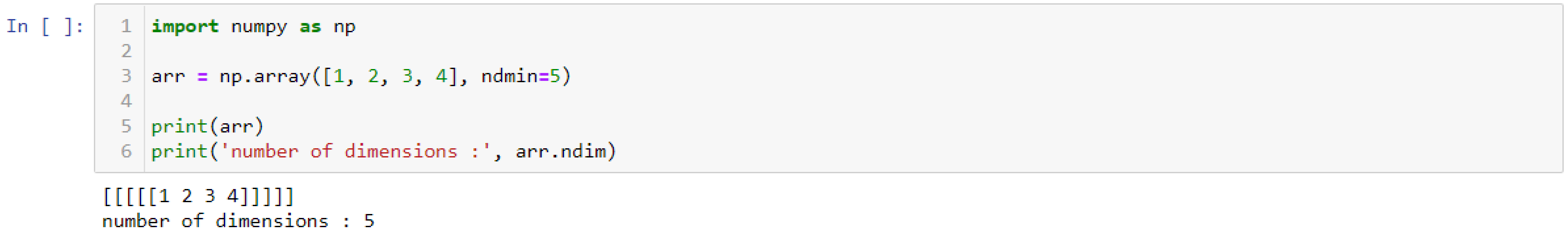


```python
import numpy as np

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

print(arr)

print(f"array shape is {arr.shape}\narray dim. is {arr.ndim}")
```

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

 [[1 2 3]
  [4 5 6]]]
array shape is (2, 2, 3)
array dim. is 3
```

5. Higher Dimensional Arrays

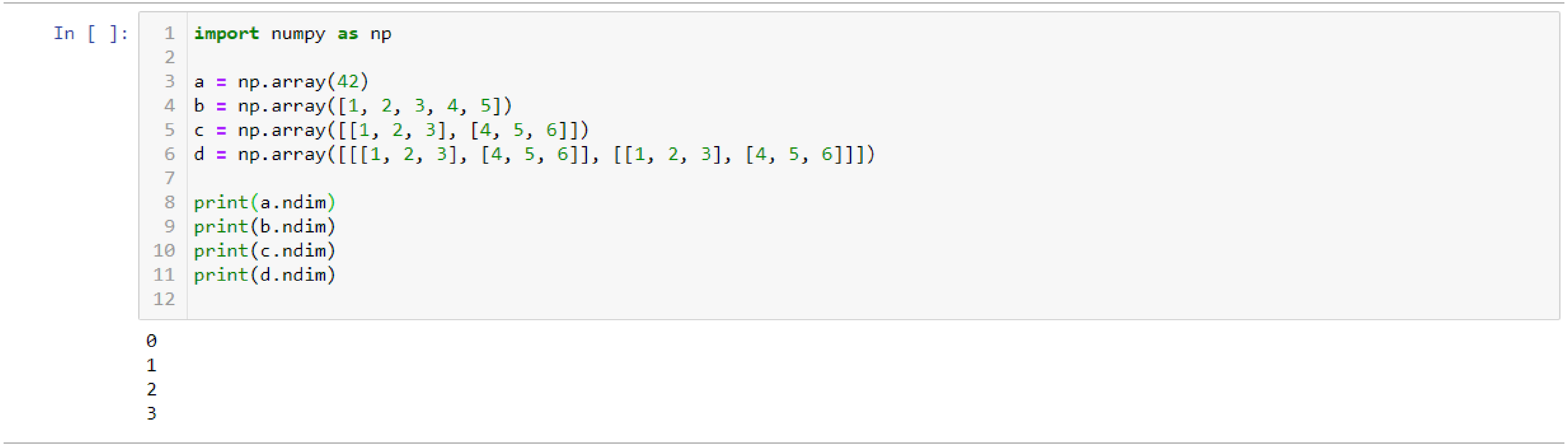
- An array can have any number of dimensions.
- When the array is created, you can define the number of dimensions by using the ndmin argument.



```python
import numpy as np

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

print(arr)
print('number of dimensions :', arr.ndim)
```

```
[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

❑ Check Number of Dimensions

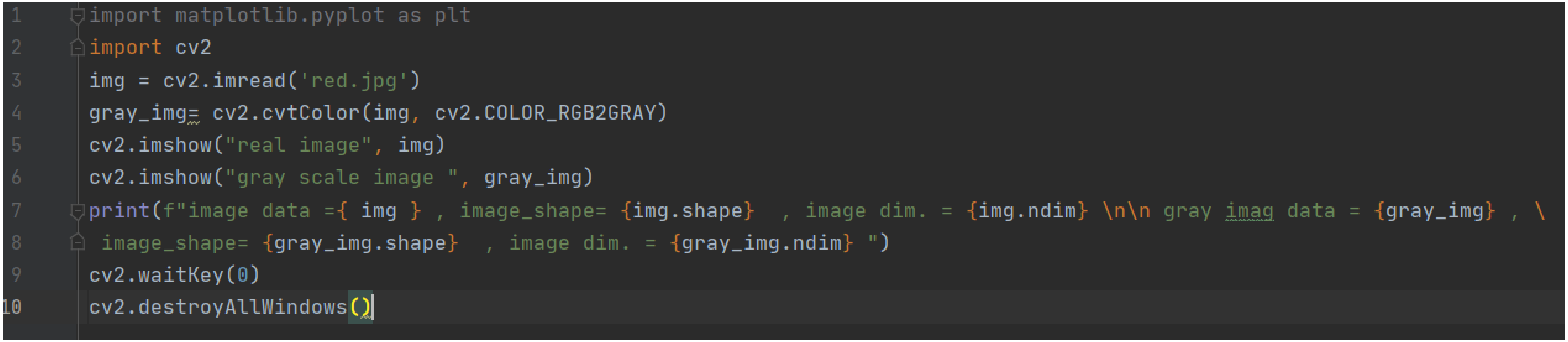


In [ ]:

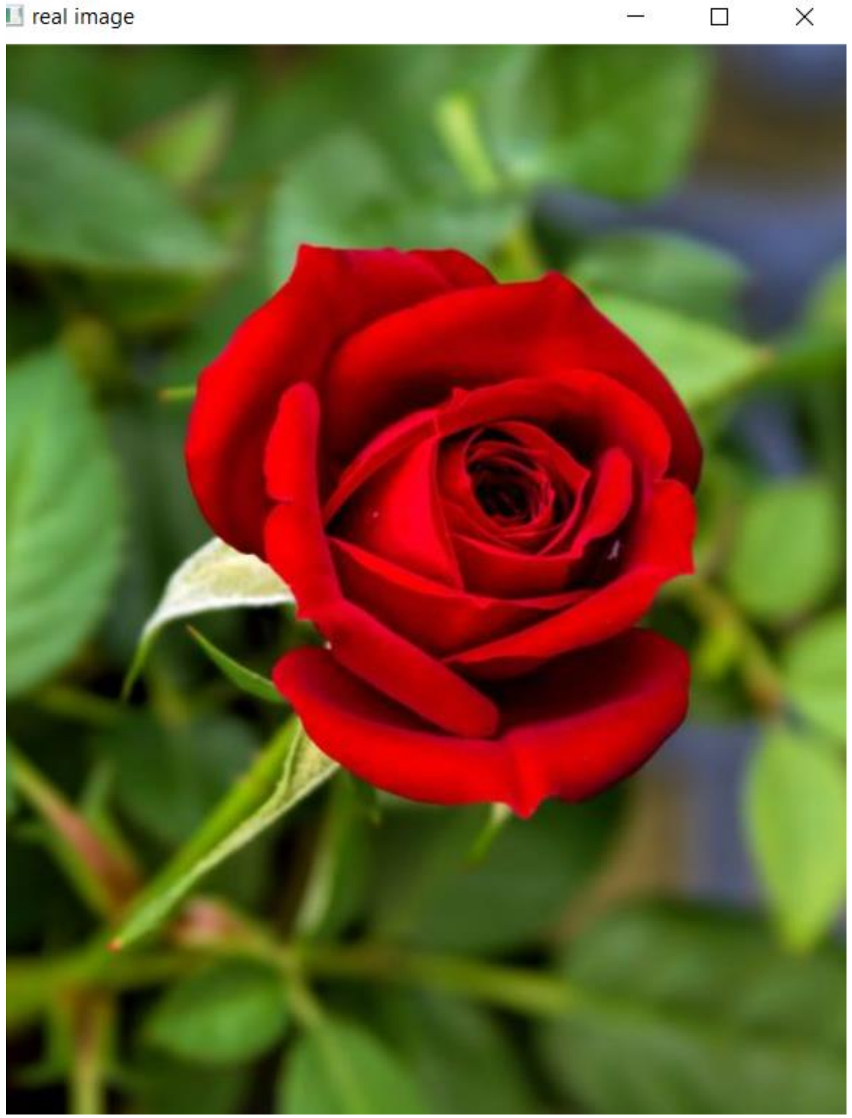
```python
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])

print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

```
0
1
2
3
```

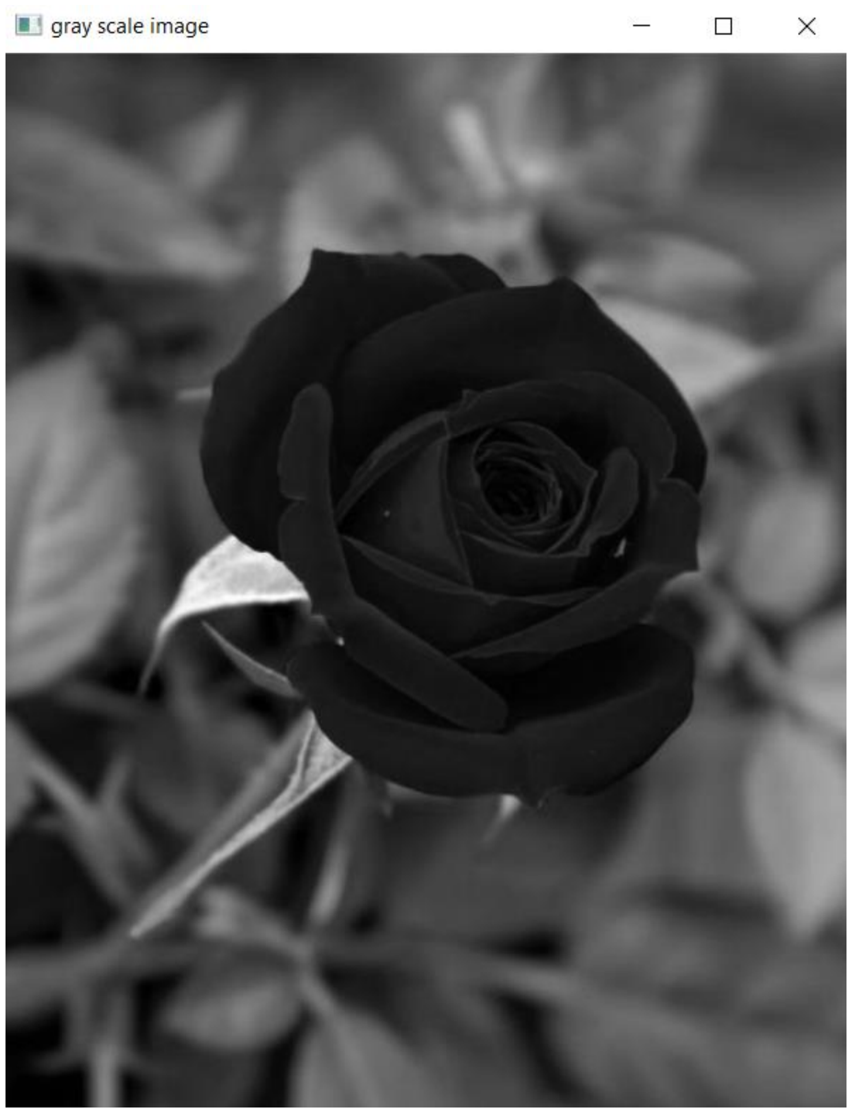
❑ 2D& 3D Array ML and data science examples



```python
import matplotlib.pyplot as plt
import cv2
img = cv2.imread('red.jpg')
gray_img= cv2.cvtColor(img, cv2.COLOR_RGB2GRAY)
cv2.imshow("real image", img)
cv2.imshow("gray scale image ", gray_img)
print(f"image data ={ img } , image_shape= {img.shape}  , image dim. = {img.ndim} \n\n gray imag data = {gray_img} , \
 image_shape= {gray_img.shape}  , image dim. = {gray_img.ndim} ")
cv2.waitKey(0)
cv2.destroyAllWindows()
```

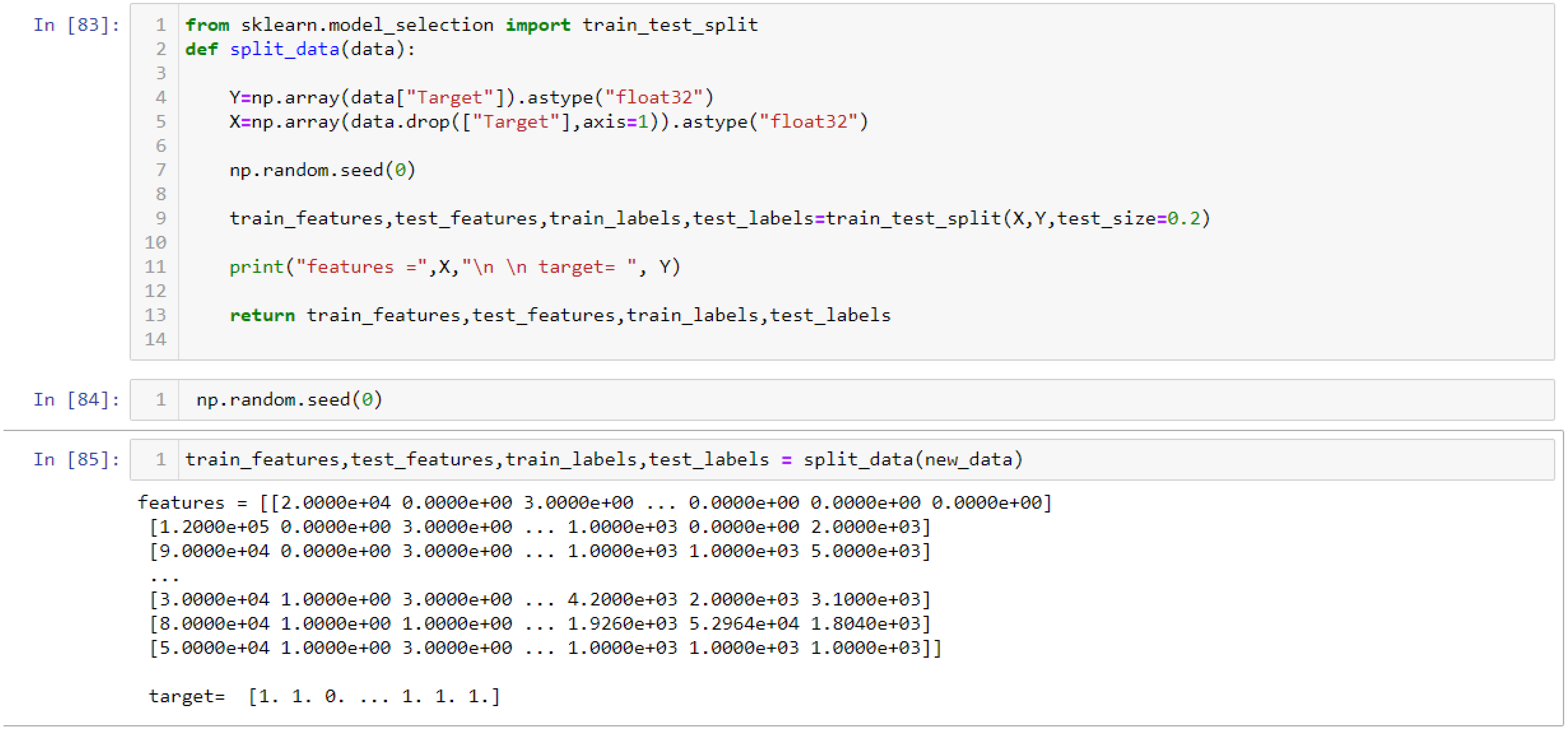


```
image data =[[[ 14  73  42]
  [ 15  74  43]
  [ 15  74  43]
  ...
  [ 40 108  77]
  [ 40 108  77]
  [ 40 108  77]]

 [[  0   0   1]
  [  0   0   1]
  [  0   0   1]
  ...
  [ 38 106  75]
  [ 38 106  75]
  [ 38 106  75]]] , image_shape= (599, 480, 3)  , image dim. = 3
```



```
gray imag data = [[ 52  53  53 ... 105 104 103]
[ 53  53  53 ... 103 102 101]
[ 53  53  53 ... 100  99  98]
...
[  1   1   1 ...  86  86  86]
[  1   1   1 ...  84  84  84]
[  0   0   0 ...  82  82  82]] ,  image_shape= (599, 480)  , image dim. = 2
```

❑ Numerical data example



```
In [83]:
from sklearn.model_selection import train_test_split
def split_data(data):

    Y=np.array(data["Target"]).astype("float32")
    X=np.array(data.drop(["Target"],axis=1)).astype("float32")

    np.random.seed(0)

    train_features,test_features,train_labels,test_labels=train_test_split(X,Y,test_size=0.2)

    print("features =",X,"\n \n target= ", Y)

    return train_features,test_features,train_labels,test_labels

```

```
In [84]:
np.random.seed(0)
```

```
In [85]:
train_features,test_features,train_labels,test_labels = split_data(new_data)
```

```
features = [[2.0000e+04 0.0000e+00 3.0000e+00 ... 0.0000e+00 0.0000e+00 0.0000e+00]
 [1.2000e+05 0.0000e+00 3.0000e+00 ... 1.0000e+03 0.0000e+00 2.0000e+03]
 [9.0000e+04 0.0000e+00 3.0000e+00 ... 1.0000e+03 1.0000e+03 5.0000e+03]
 ...
 [3.0000e+04 1.0000e+00 3.0000e+00 ... 4.2000e+03 2.0000e+03 3.1000e+03]
 [8.0000e+04 1.0000e+00 1.0000e+00 ... 1.9260e+03 5.2964e+04 1.8040e+03]
 [5.0000e+04 1.0000e+00 3.0000e+00 ... 1.0000e+03 1.0000e+03 1.0000e+03]]

 target=  [1. 1. 0. ... 1. 1. 1.]
```

❑ Create 1D / 2D Numpy Array filled with ones (1’s)

- **numpy.ones()**

Python’s Numpy module provides a function to create a numpy array of given shape & type and filled with 1’s

numpy.ones(shape, dtype=float, order='C')

Arguments:

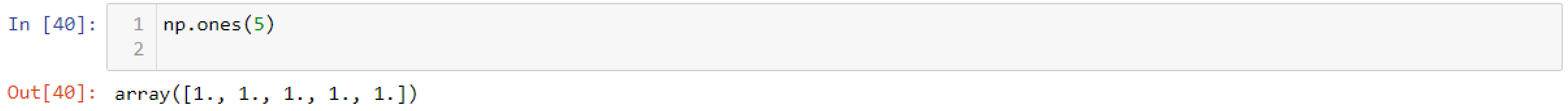
- **shape:** Shape of the numpy array. Single integer or sequence of integers.
- **dtype:** (Optional) Data type of elements. Default is float64.
- **order:** (Optional) Order in which data is stored in multi-dimension array i.e. in row major(‘F’) or column major (‘C’). Default is ‘C’.

Returns:

- It returns a numpy array of given shape but filled with ones.

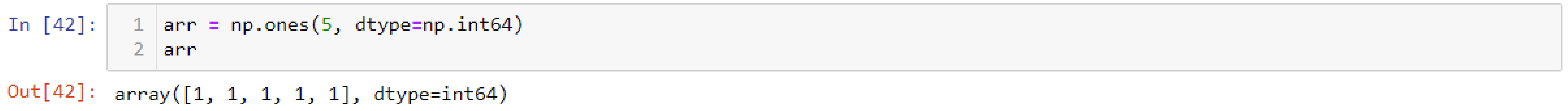
1. Create 1D Numpy Array of given length and filled with ones

Suppose we want to create a numpy array of five ones (1s). For that we need to call the numpy.ones() function with argument 5.



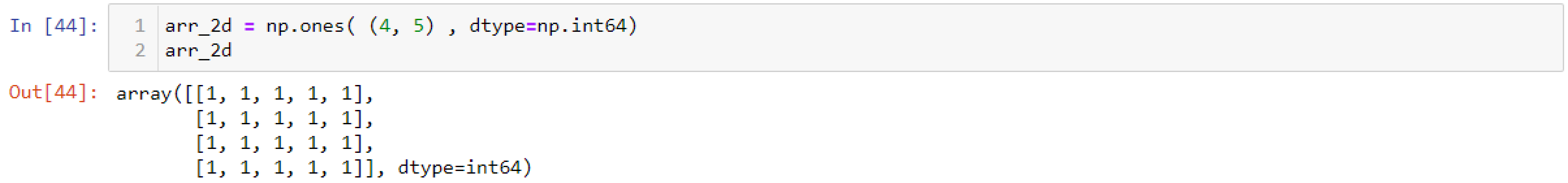
```
In [40]: np.ones(5)

Out[40]: array([1., 1., 1., 1., 1.])
```

- Create Numpy array with ones of integer data type



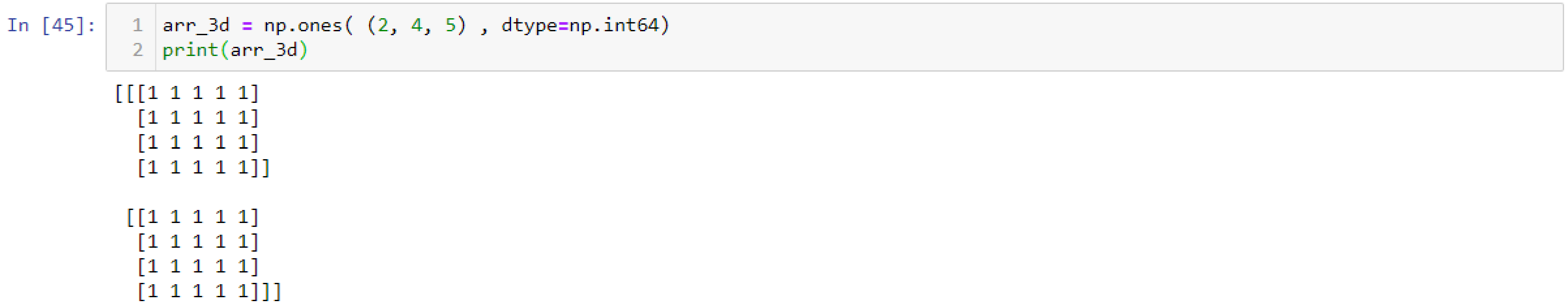
```
In [42]: arr = np.ones(5, dtype=np.int64)
         arr

Out[42]: array([1, 1, 1, 1, 1], dtype=int64)
```

2. Create two dimensional (2D) Numpy Array of ones



```
In [44]:  arr_2d = np.ones( (4, 5) , dtype=np.int64)
           arr_2d

Out[44]: array([[1, 1, 1, 1, 1],
                [1, 1, 1, 1, 1],
                [1, 1, 1, 1, 1],
                [1, 1, 1, 1, 1]], dtype=int64)
```

3. Create 3D Numpy Array filled with ones



```
In [45]:  arr_3d = np.ones( (2, 4, 5) , dtype=np.int64)
           print(arr_3d)

[[[1 1 1 1 1]
  [1 1 1 1 1]
  [1 1 1 1 1]
  [1 1 1 1 1]]

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

❑ The standard data types of NumPy

Data Types in Python

By default Python have these data types:

1. strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
2. integer - used to represent integer numbers. e.g. -1, -2, -3
3. float - used to represent real numbers. e.g. 1.2, 42.42
4. boolean - used to represent True or False.
5. complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

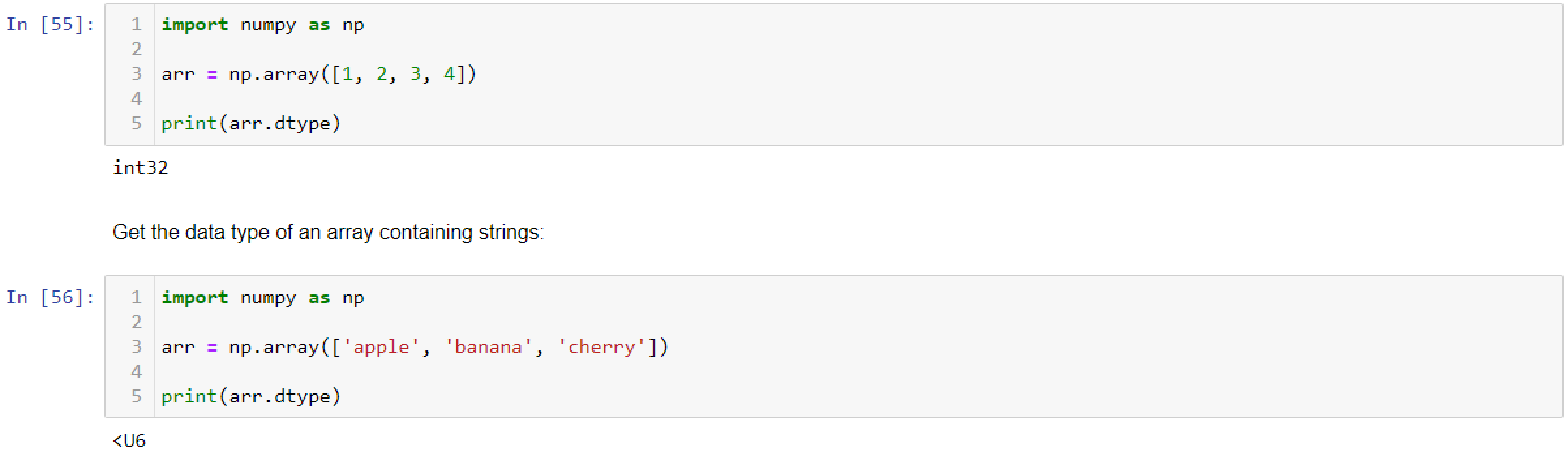
Below is a list of all data types in NumPy and the characters used to represent them.

i – integer	b - boolean
u - unsigned integer	f - float
c - complex float	m - timedelta
M – datetime	O - object
S – string	U - unicode string
V - fixed chunk of memory for other type (void)	

https://numpy.org/doc/1.20/user/basics.types.html

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:



In [55]:
```python
import numpy as np

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

print(arr.dtype)
```

```
int32
```

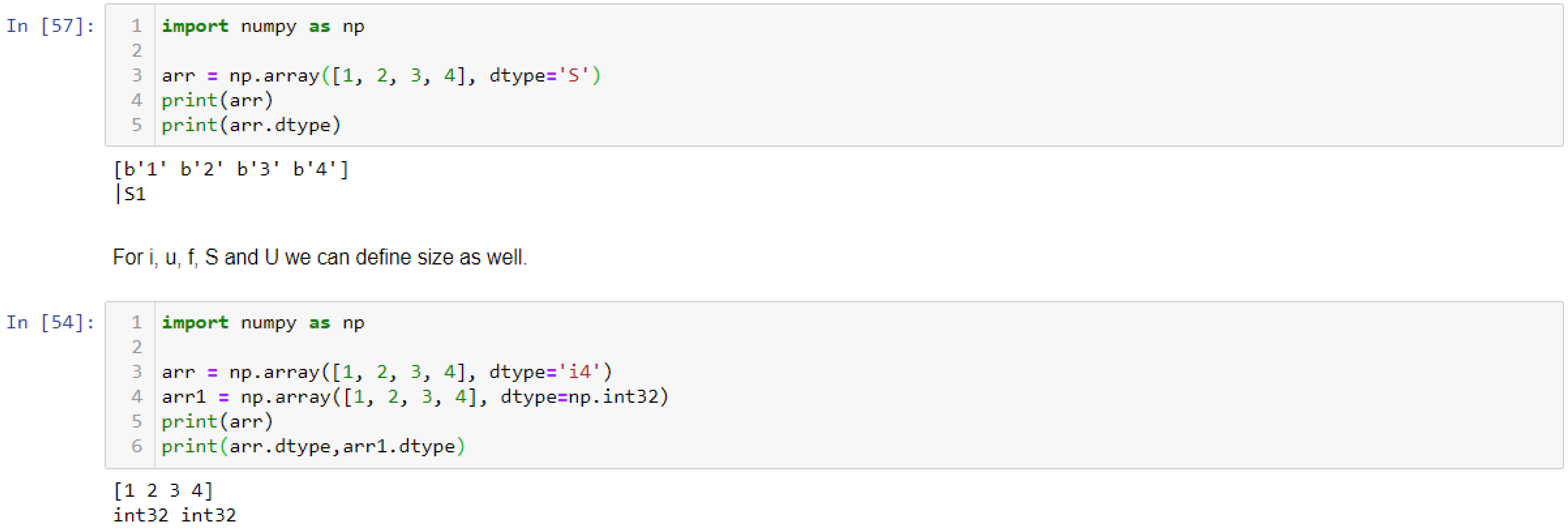
Get the data type of an array containing strings:

In [56]:
```python
import numpy as np

arr = np.array(['apple', 'banana', 'cherry'])

print(arr.dtype)
```

```
<U6
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements



In [57]:

```python
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

```
[b'1' b'2' b'3' b'4']
|S1
```

For i, u, f, S and U we can define size as well.

In [54]:

```python
import numpy as np

arr = np.array([1, 2, 3, 4], dtype='i4')
arr1 = np.array([1, 2, 3, 4], dtype=np.int32)
print(arr)
print(arr.dtype,arr1.dtype)
```

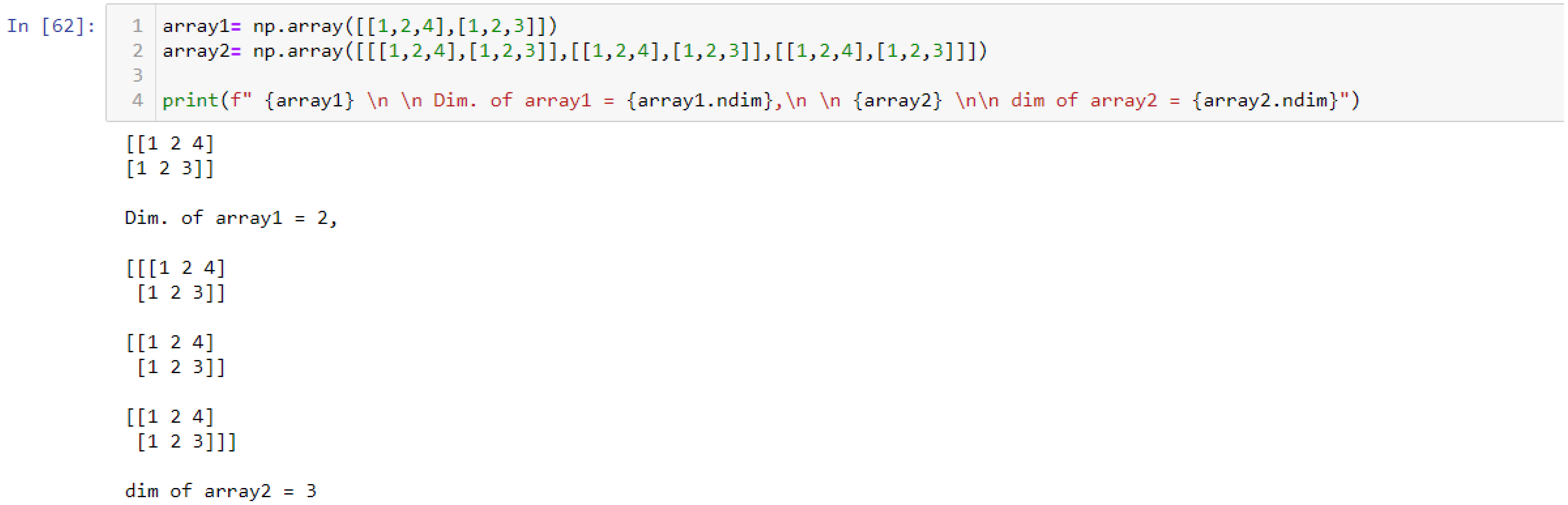
```
[1 2 3 4]
int32 int32
```

❑ Basics attribute of array

Basic Array Attributes

Armed with our understanding of multidimensional NumPy arrays, we now look at methods for programmatically inspecting an array’s attributes (e.g. its dimensionality). It is especially important to understand what an array’s “shape” is.

1. ndarray.ndim:

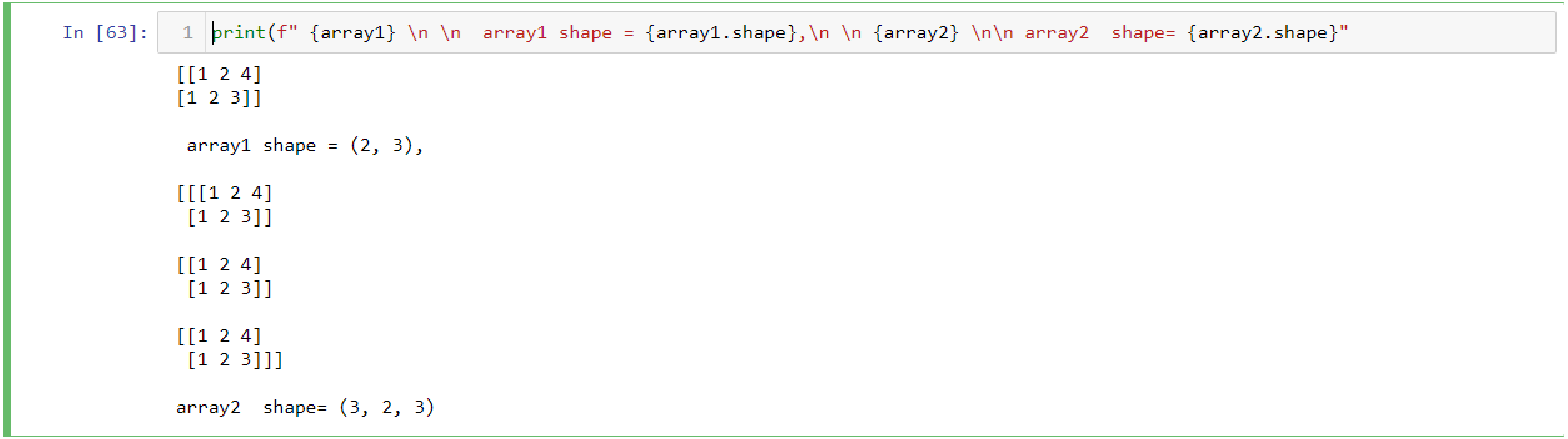


```
In [62]:
array1= np.array([[1,2,4],[1,2,3]])
array2= np.array([[[1,2,4],[1,2,3]],[[1,2,4],[1,2,3]],[[1,2,4],[1,2,3]]])

print(f" {array1} \n \n Dim. of array1 = {array1.ndim},\n \n {array2} \n\n dim of array2 = {array2.ndim}")
```

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

Dim. of array1 = 2,

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

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

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

dim of array2 = 3
```

2. ndarray.shape:

A tuple of integers indicating the number of elements that are stored along each dimension of the array. For a 2D-array with NN rows and MM columns, shape will be (N,M)(N,M). The length of this shape-tuple is therefore equal to the number of dimensions of the array.



```
In [63]:  print(f" {array1} \n \n  array1 shape = {array1.shape},\n \n {array2} \n\n array2  shape= {array2.shape}")
```

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

 array1 shape = (2, 3),

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

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

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

array2  shape= (3, 2, 3)
```

3. ndarray.size:

The total number of elements of the array. This is equal to the product of the elements of the array’s shape

```
In [68]:  print(f" {array1} \n \n  array1 size = {array1.size},\n \n {array2} \n\n array2  size= {array2.size}")
```

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

 array1 size = 6,

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

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

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

array2  size= 18
```

4. ndarray.dtype:

An object describing the data type of the elements in the array.

```
In [69]:  print(f" {array1} \n \n  array2 data type = {array1.dtype},\n \n {array2} \n\n array2 data type = {array2.dtype}")
```

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

 array2 data type = int32,

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

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

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

array2 data type = int32
```

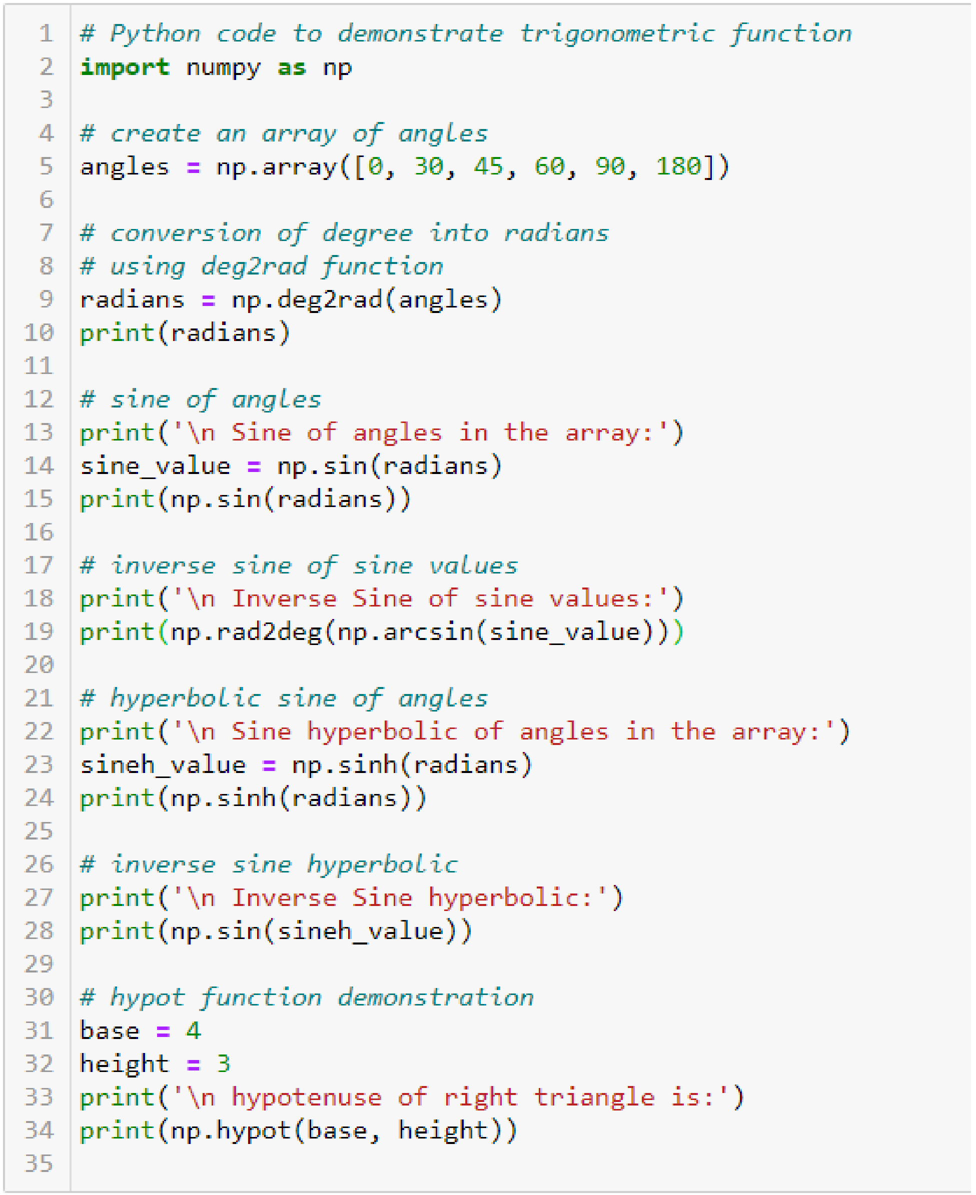
❑ The basic universal functions of NumPy

Universal functions in Numpy are simple mathematical functions. It is just a term that we gave to mathematical functions in the Numpy library. Numpy provides various universal functions that cover a wide variety of operations.
These functions include standard trigonometric functions, functions for arithmetic operations, handling complex numbers, statistical functions, etc.

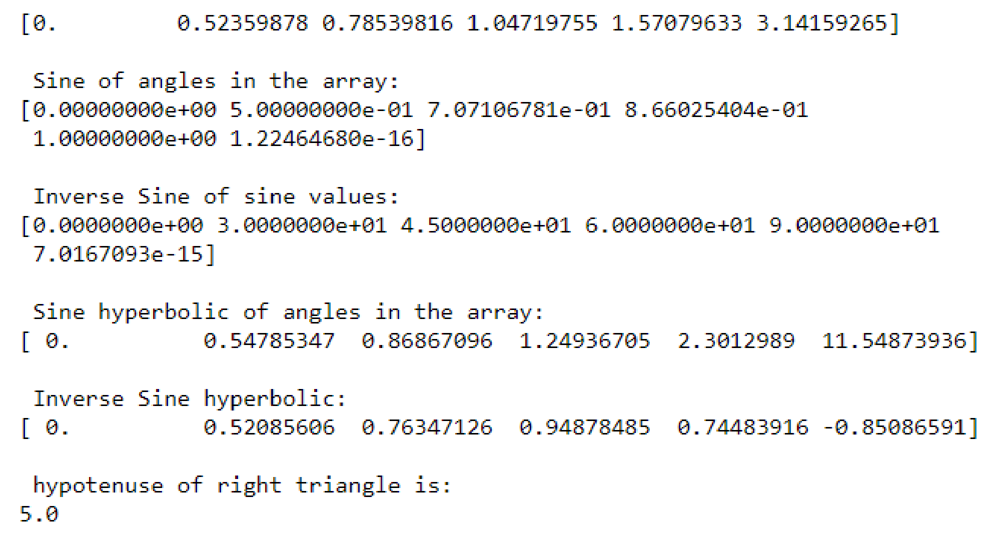
Trigonometric functions:

These functions work on radians, so angles need to be converted to radians by multiplying by pi/180. Only then we can call trigonometric functions. They take an array as input arguments. It includes functions like

Function	Description
sin, cos, tan	compute sine, cosine and tangent of angles
arcsin, arccos, arctan	calculate inverse sine, cosine and tangent
hypot	calculate hypotenuse of given right triangle
sinh, cosh, tanh	compute hyperbolic sine, cosine and tangent
arcsinh, arccosh, arctanh	compute inverse hyperbolic sine, cosine and tangent
deg2rad	convert degree into radians
rad2deg	convert radians into degree



```python
# Python code to demonstrate trigonometric function
import numpy as np

# create an array of angles
angles = np.array([0, 30, 45, 60, 90, 180])

# conversion of degree into radians
# using deg2rad function
radians = np.deg2rad(angles)
print(radians)

# sine of angles
print('\n Sine of angles in the array:')
sine_value = np.sin(radians)
print(np.sin(radians))

# inverse sine of sine values
print('\n Inverse Sine of sine values:')
print(np.rad2deg(np.arcsin(sine_value)))

# hyperbolic sine of angles
print('\n Sine hyperbolic of angles in the array:')
sineh_value = np.sinh(radians)
print(np.sinh(radians))

# inverse sine hyperbolic
print('\n Inverse Sine hyperbolic:')
print(np.sin(sineh_value))

# hypot function demonstration
base = 4
height = 3
print('\n hypotenuse of right triangle is:')
print(np.hypot(base, height))

```

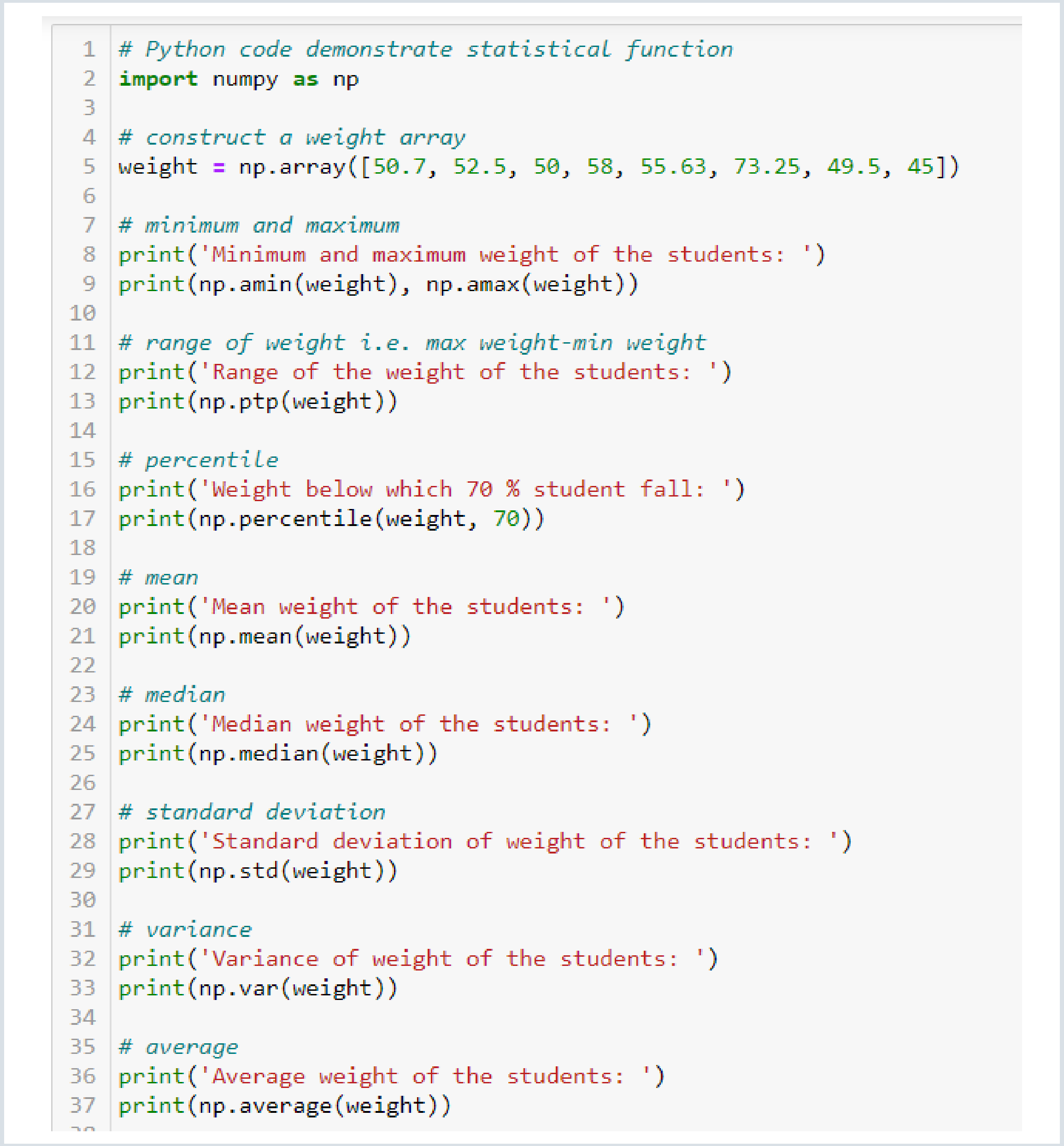


```
[0.         0.52359878 0.78539816 1.04719755 1.57079633 3.14159265]

 Sine of angles in the array:
[0.00000000e+00 5.00000000e-01 7.07106781e-01 8.66025404e-01
 1.00000000e+00 1.22464680e-16]

 Inverse Sine of sine values:
[0.0000000e+00 3.0000000e+01 4.5000000e+01 6.0000000e+01 9.0000000e+01
 7.0167093e-15]

 Sine hyperbolic of angles in the array:
[ 0.          0.54785347  0.86867096  1.24936705  2.3012989  11.54873936]

 Inverse Sine hyperbolic:
[ 0.          0.52085606  0.76347126  0.94878485  0.74483916 -0.85086591]

 hypotenuse of right triangle is:
5.0
```

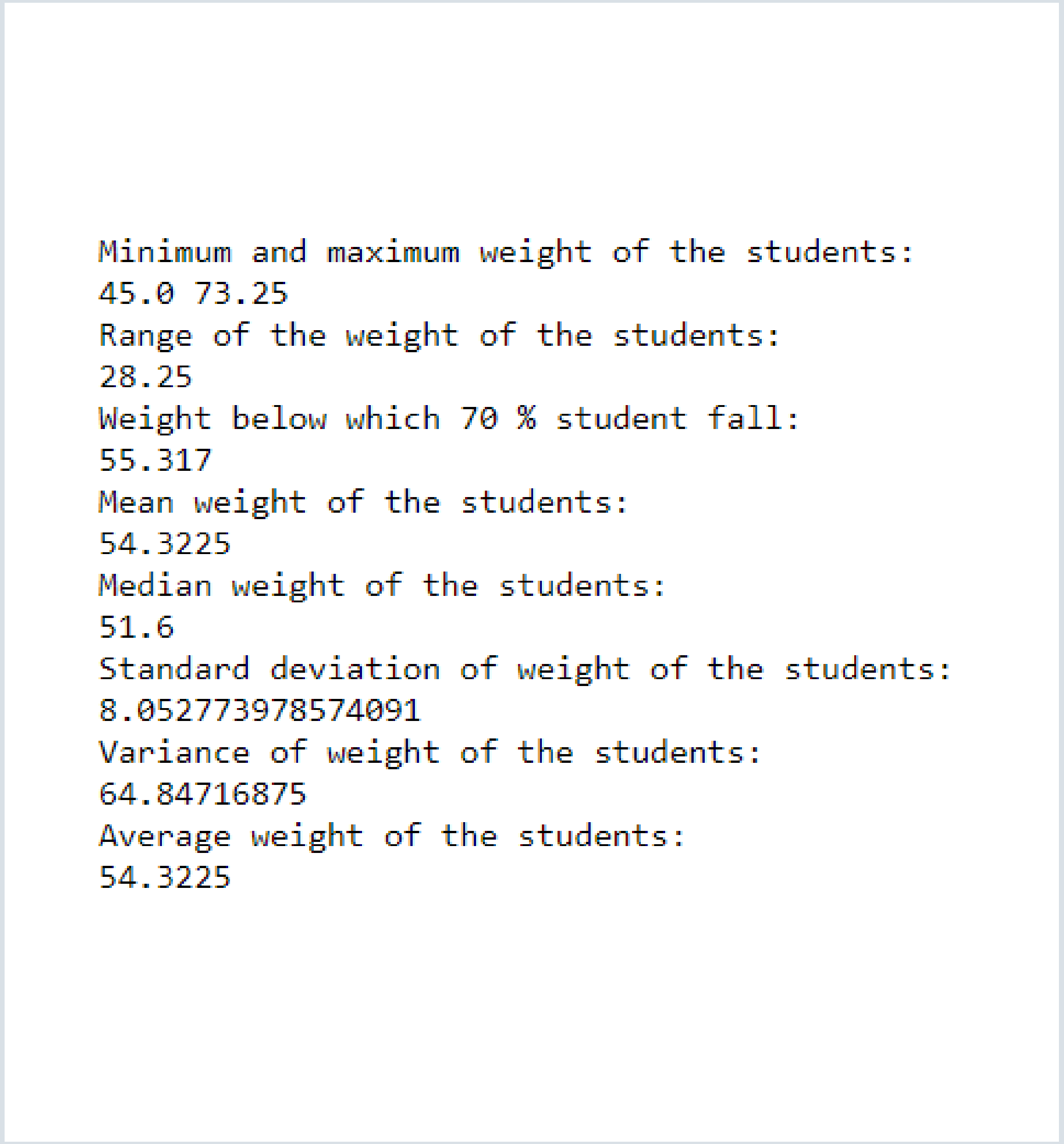
Statistical functions:

These functions are used to calculate mean, median, variance, minimum of array elements. It includes functions like

Function	Description
amin, amax	returns minimum or maximum of an array or along an axis
ptp	returns range of values (maximum-minimum) of an array or along an axis
percentile(a, p, axis)	calculate pth percentile of array or along specified axis
median	compute median of data along specified axis
mean	compute mean of data along specified axis
std	compute standard deviation of data along specified axis
var	compute variance of data along specified axis
average	compute average of data along specified axis



```python
# Python code demonstrate statistical function
import numpy as np

# construct a weight array
weight = np.array([50.7, 52.5, 50, 58, 55.63, 73.25, 49.5, 45])

# minimum and maximum
print('Minimum and maximum weight of the students: ')
print(np.amin(weight), np.amax(weight))

# range of weight i.e. max weight-min weight
print('Range of the weight of the students: ')
print(np.ptp(weight))

# percentile
print('Weight below which 70 % student fall: ')
print(np.percentile(weight, 70))

# mean
print('Mean weight of the students: ')
print(np.mean(weight))

# median
print('Median weight of the students: ')
print(np.median(weight))

# standard deviation
print('Standard deviation of weight of the students: ')
print(np.std(weight))

# variance
print('Variance of weight of the students: ')
print(np.var(weight))

# average
print('Average weight of the students: ')
print(np.average(weight))
```

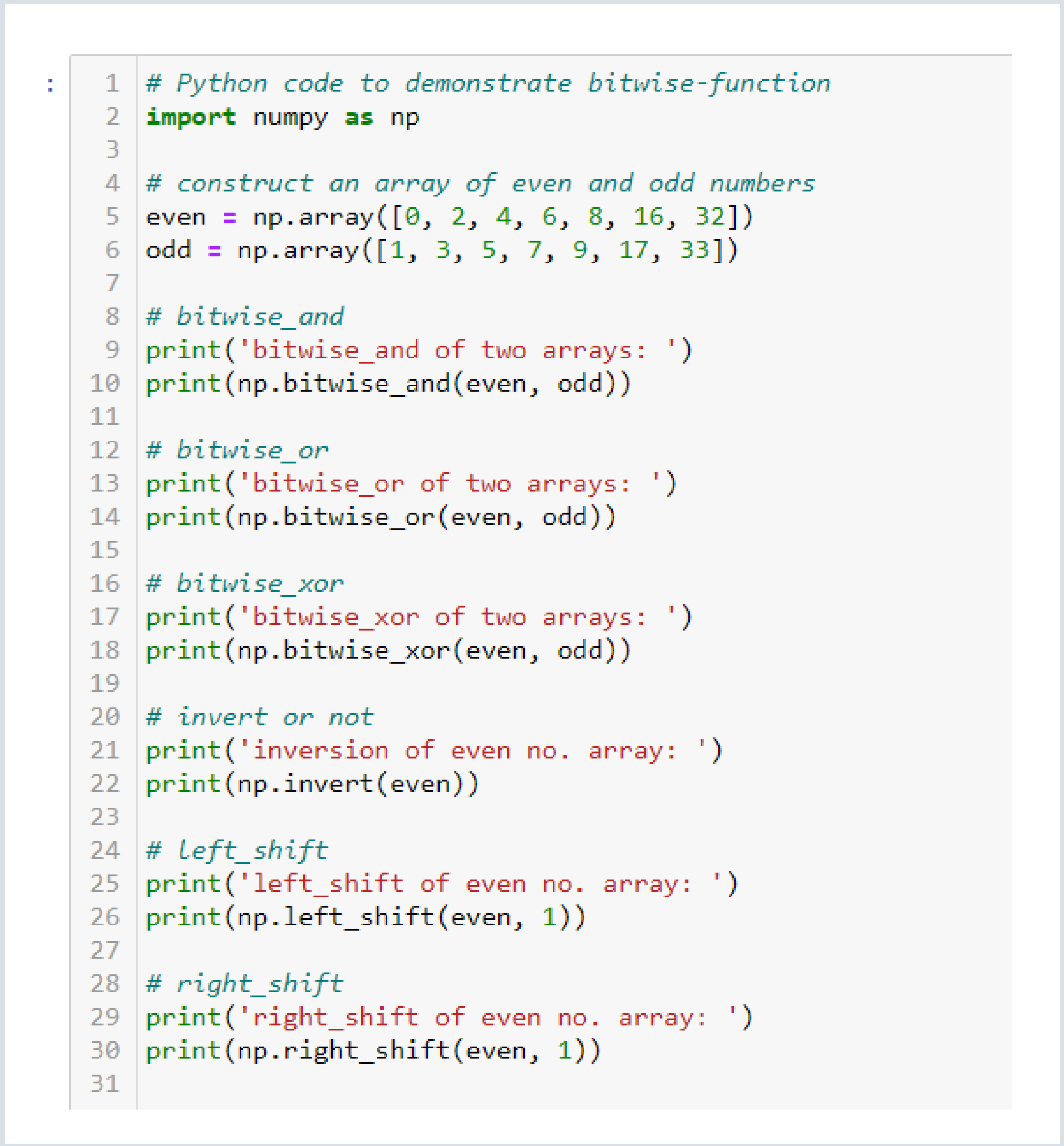


```
Minimum and maximum weight of the students:
45.0 73.25
Range of the weight of the students:
28.25
Weight below which 70 % student fall:
55.317
Mean weight of the students:
54.3225
Median weight of the students:
51.6
Standard deviation of weight of the students:
8.052773978574091
Variance of weight of the students:
64.84716875
Average weight of the students:
54.3225
```

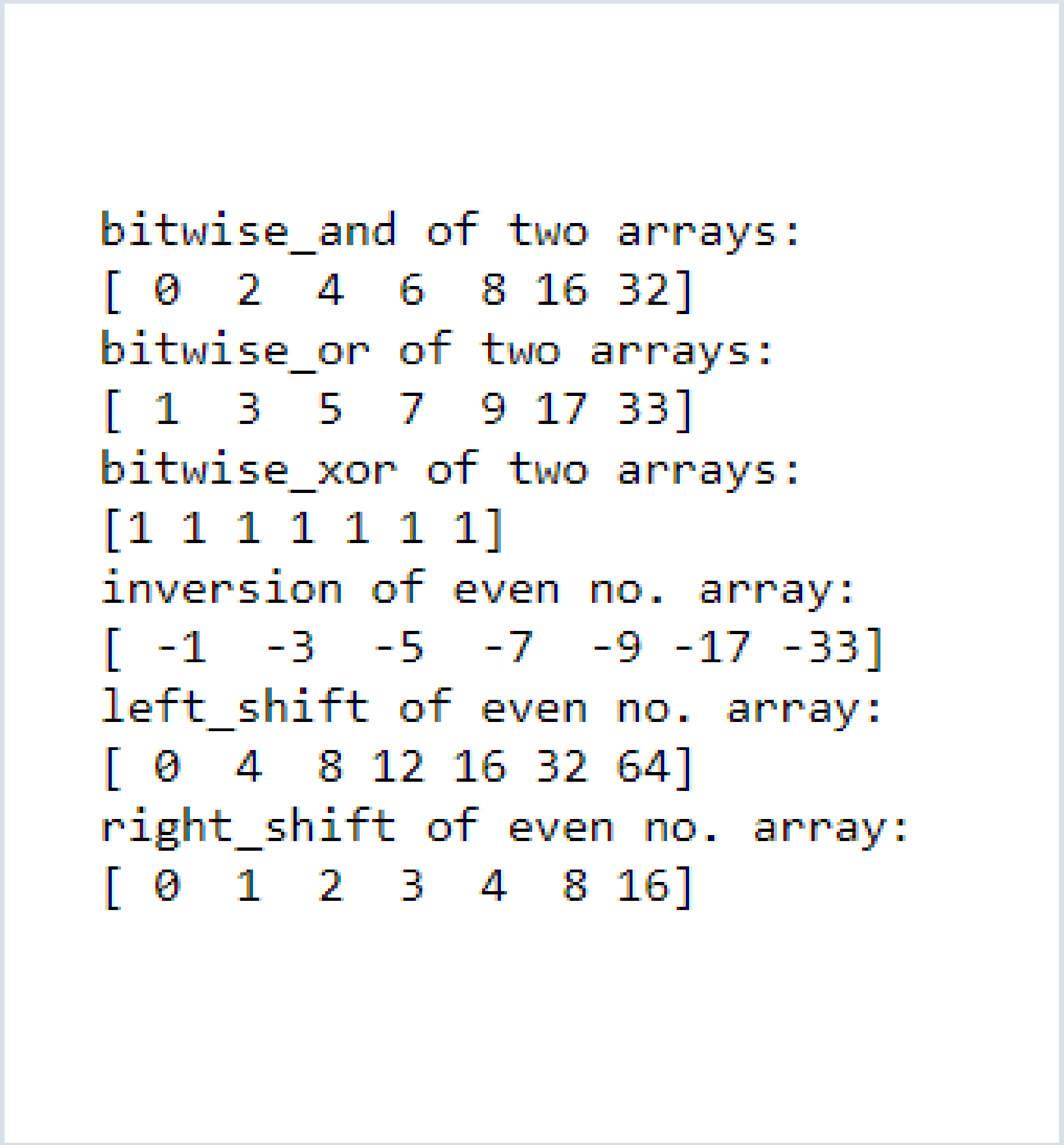
Bit-twiddling functions:

These functions accept integer values as input arguments and perform bitwise operations on binary representations of those integers. It include functions like-

Function	Description
bitwise_and	performs bitwise and operation on two array elements
bitwies_or	performs bitwise or operation on two array elements
bitwise_xor	performs bitwise xor operation on two array elements
invert	performs bitwise inversion of an array elements
left_shift	shift the bits of elements to left
right_shift	shift the bits of elements to left



```python
# Python code to demonstrate bitwise-function
import numpy as np

# construct an array of even and odd numbers
even = np.array([0, 2, 4, 6, 8, 16, 32])
odd = np.array([1, 3, 5, 7, 9, 17, 33])

# bitwise_and
print('bitwise_and of two arrays: ')
print(np.bitwise_and(even, odd))

# bitwise_or
print('bitwise_or of two arrays: ')
print(np.bitwise_or(even, odd))

# bitwise_xor
print('bitwise_xor of two arrays: ')
print(np.bitwise_xor(even, odd))

# invert or not
print('inversion of even no. array: ')
print(np.invert(even))

# left_shift
print('left_shift of even no. array: ')
print(np.left_shift(even, 1))

# right_shift
print('right_shift of even no. array: ')
print(np.right_shift(even, 1))

```



```
bitwise_and of two arrays:
[ 0  2  4  6  8 16 32]
bitwise_or of two arrays:
[ 1  3  5  7  9 17 33]
bitwise_xor of two arrays:
[1 1 1 1 1 1 1]
inversion of even no. array:
[ -1  -3  -5  -7  -9 -17 -33]
left_shift of even no. array:
[ 0  4  8 12 16 32 64]
right_shift of even no. array:
[ 0  1  2  3  4  8 16]
```

Arithmetic functions:

Operator	Equivalent ufunc	Description
+	np.add	Addition (e.g., 1 + 1 = 2)
-	np.subtract	Subtraction (e.g., 3 - 2 = 1)
-	np.negative	Unary negation (e.g., -2)
*	np.multiply	Multiplication (e.g., 2 * 3 = 6)
/	np.divide	Division (e.g., 3 / 2 = 1.5)
//	np.floor_divide	Floor division (e.g., 3 // 2 = 1)
**	np.power	Exponentiation (e.g., 2 ** 3 = 8)
%	np.mod	Modulus/remainder (e.g., 9 % 4 = 1)

❑ Boolean indexing of arrays, slice arrays, reshape arrays

1. NumPy Array Indexing

You can access an array element by referring to its index number.
The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

```python
import numpy as np

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

print(arr[0])
```

```
1
```

```python
import numpy as np

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

print(arr[1])
```

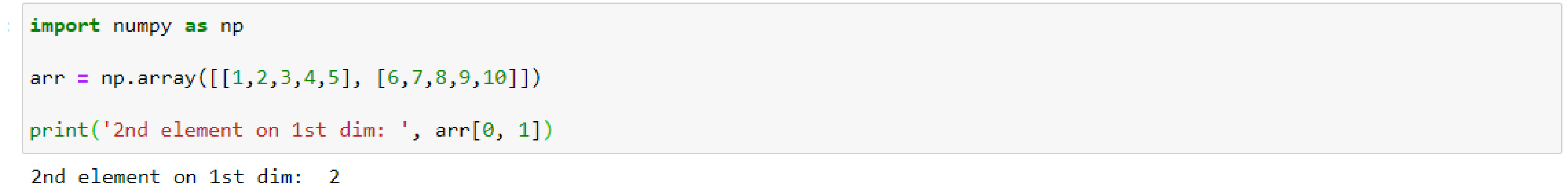
```
2
```

```python
import numpy as np

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

print(arr[2] + arr[3])
```

```
7
```

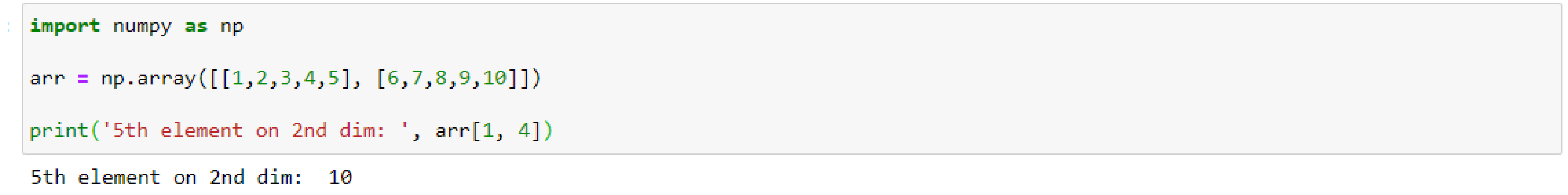
Access 2-D Arrays: *array[row index, column index]*

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.



```python
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('2nd element on 1st dim: ', arr[0, 1])
```

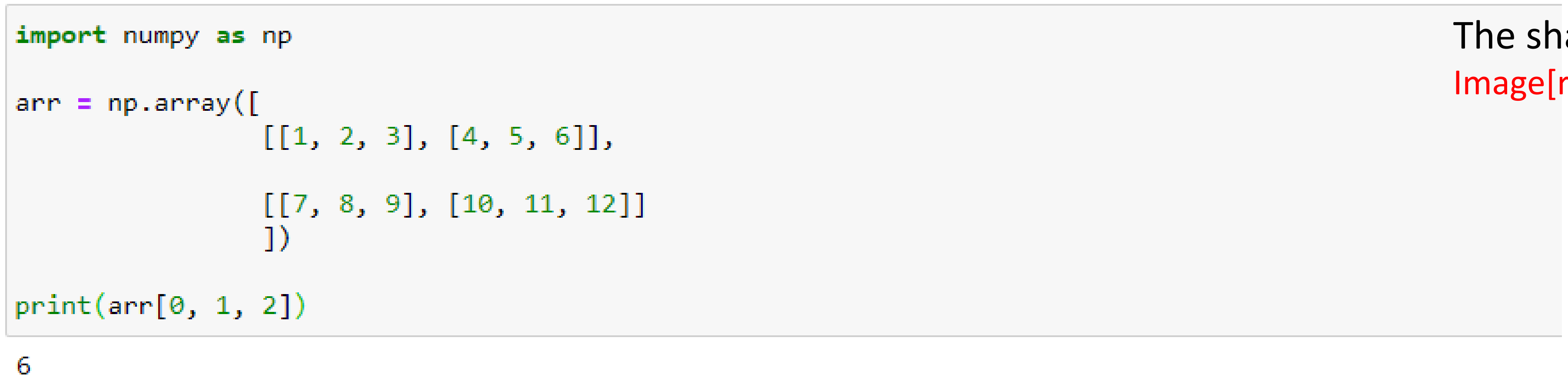
```
2nd element on 1st dim:  2
```



```python
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('5th element on 2nd dim: ', arr[1, 4])
```

```
5th element on 2nd dim:  10
```

Access 3-D Arrays *array[channel index ,row index, column index]*

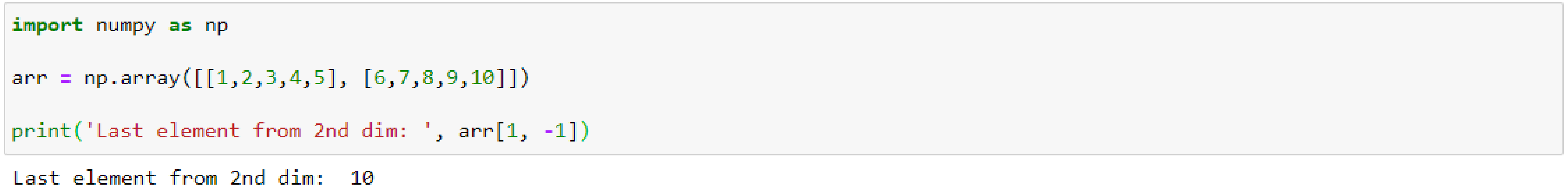


```python
import numpy as np

arr = np.array([
                [[1, 2, 3], [4, 5, 6]],

                [[7, 8, 9], [10, 11, 12]]
                ])

print(arr[0, 1, 2])
```

```
6
```

Note: if you read image data
The shape terms reversed.
Image[rows,columns,channels]

Negative Indexing



```python
import numpy as np

arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])

print('Last element from 2nd dim: ', arr[1, -1])
```

```
Last element from 2nd dim:  10
```

2. NumPy Array Slicing

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: **[start:end].**
We can also define the step, like this: [start:end:step].

- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step its considered 1

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np

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

print(arr[1:5])
```

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

Slice elements from index 4 to the end of the array:

```
import numpy as np

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

print(arr[4:])
```

```
[5 6 7]
```

Slice elements from the beginning to index 4 (not included):

```
import numpy as np

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

print(arr[:4])
```

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

Negative Slicing

Slice from the index 3 from the end to index 1 from the end:

```
import numpy as np

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

print(arr[-3:-1])
```

```
[5 6]
```

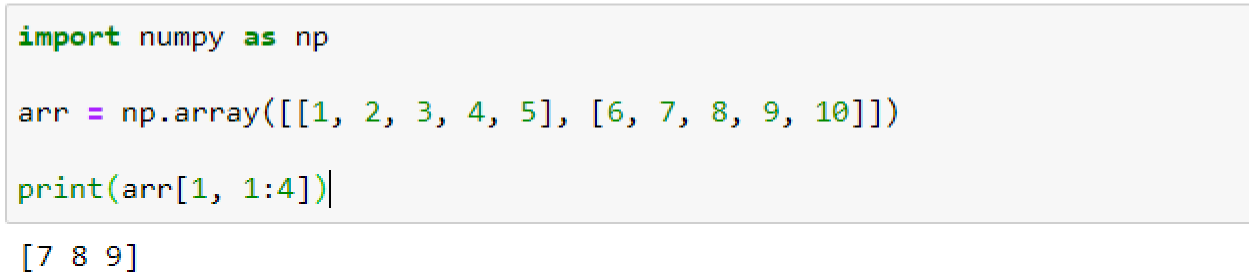
Use the step value to determine the step of the slicing:

```
import numpy as np

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

print(arr[::-1])
print(arr[::-2])
print(arr[-1:-8:-1])
print(arr[-1:-4:-2])
```

```
[7 6 5 4 3 2 1]
[7 5 3 1]
[7 6 5 4 3 2 1]
[7 5]
```

Slicing 2-D Arrays

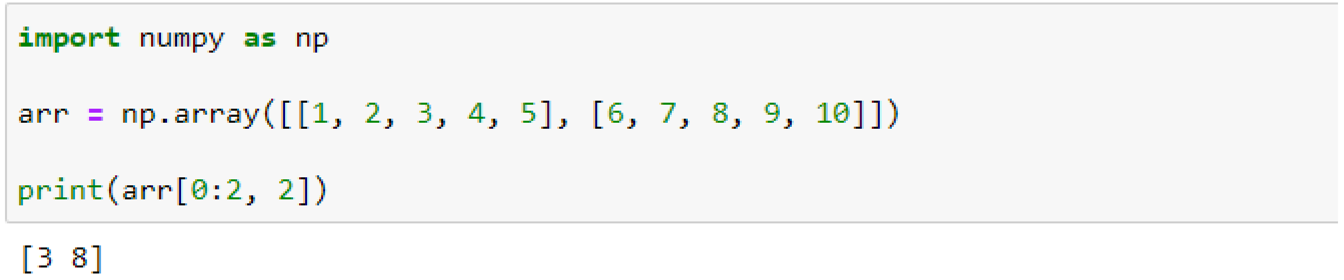
From the second element, slice elements from index 1 to index 4 (not included):



```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

```
[7 8 9]
```

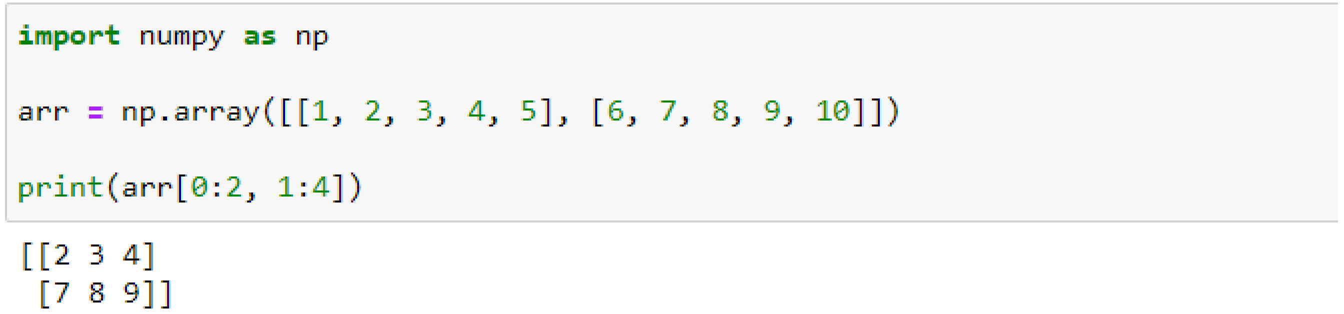
From both elements, return index 2:



```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

```
[3 8]
```

From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:



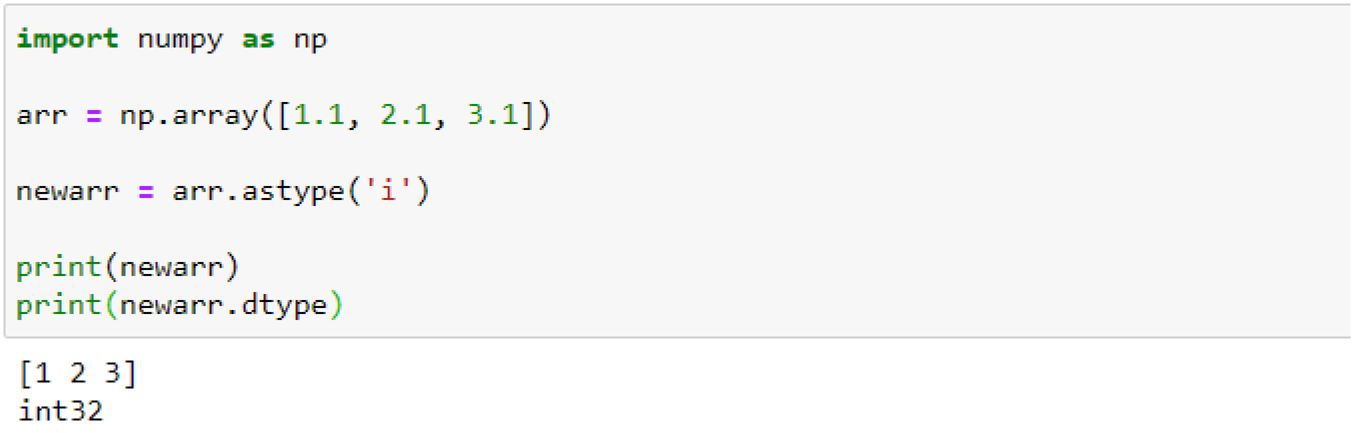
```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])

print(arr[0:2, 1:4])
```

```
[[2 3 4]
 [7 8 9]]
```

3. Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method. The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.
The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

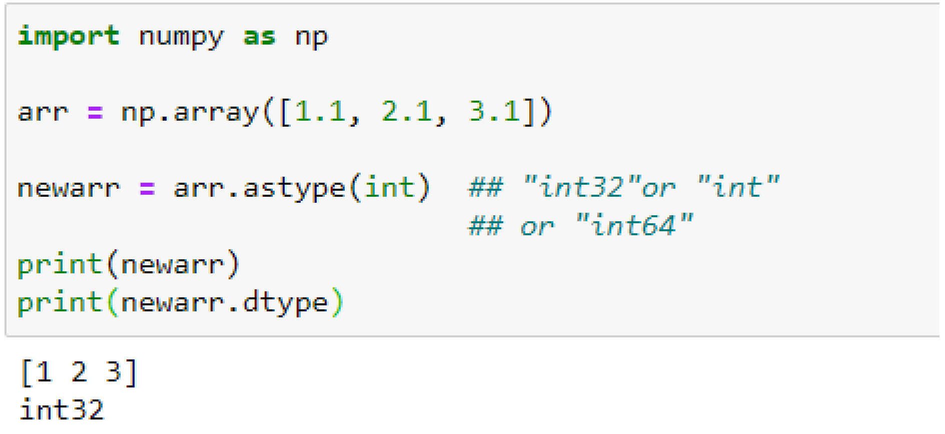
Change data type from float to integer by using **'i'** as parameter value:



```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

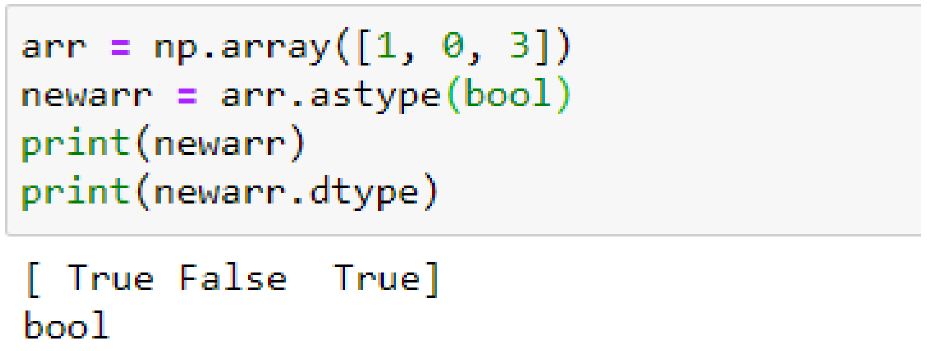
Change data type from float to **integer** by
using **int** as parameter value:



```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])

newarr = arr.astype(int)  ## "int32"or "int"
                          ## or "int64"
print(newarr)
print(newarr.dtype)
```

```
[1 2 3]
int32
```

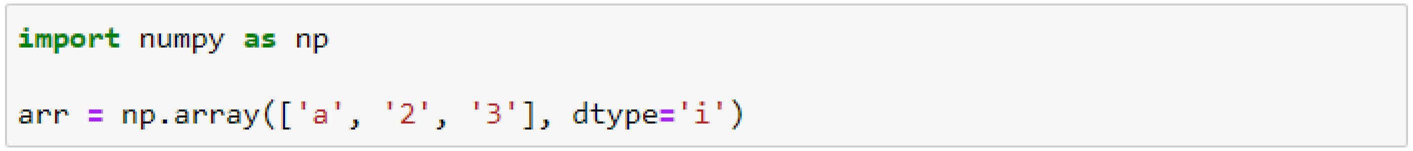
Change data type from integer to **Boolean**



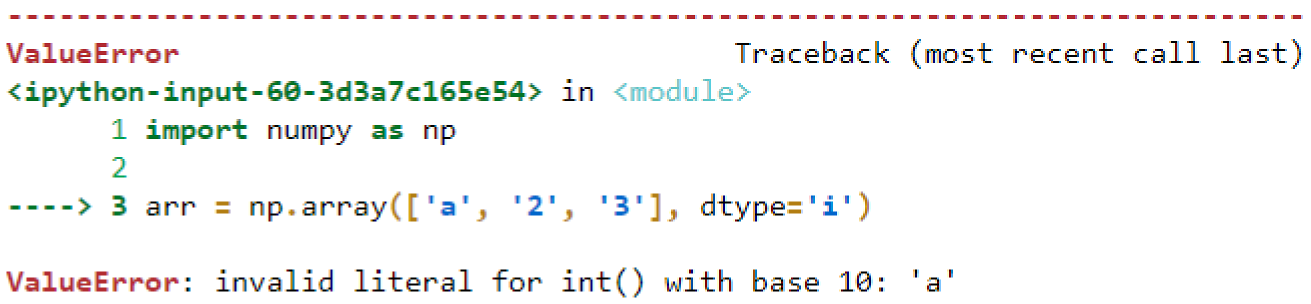
```
arr = np.array([1, 0, 3])
newarr = arr.astype(bool)
print(newarr)
print(newarr.dtype)
```

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

What is the result of code below ?



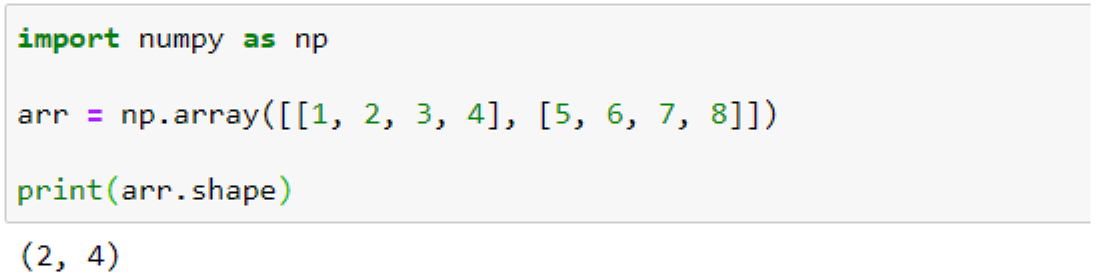
```
import numpy as np

arr = np.array(['a', '2', '3'], dtype='i')
```



```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-60-3d3a7c165e54> in <module>
      1 import numpy as np
      2
----> 3 arr = np.array(['a', '2', '3'], dtype='i')

ValueError: invalid literal for int() with base 10: 'a'
```

4. NumPy Array Shape

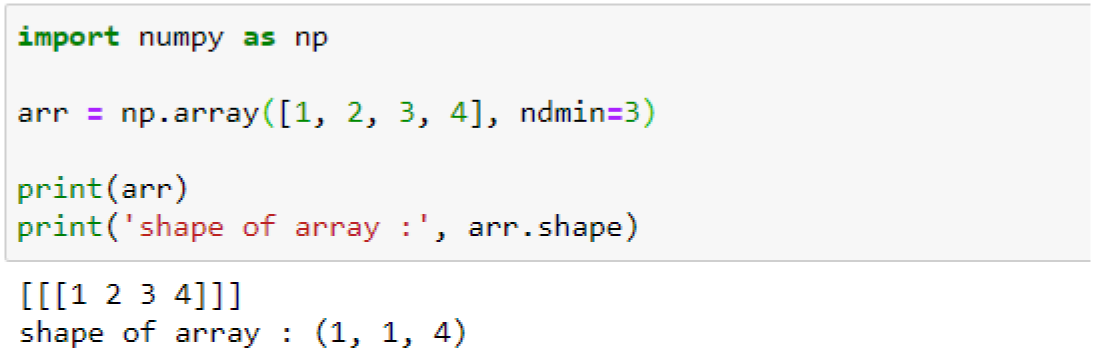
The shape of an array is the number of elements in each dimension. NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.



```python
import numpy as np

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

print(arr.shape)
```

```
(2, 4)
```

Create an array with 3 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:



```python
import numpy as np

arr = np.array([1, 2, 3, 4], ndmin=3)

print(arr)
print('shape of array :', arr.shape)
```

```
[[[1 2 3 4]]]
shape of array : (1, 1, 4)
```

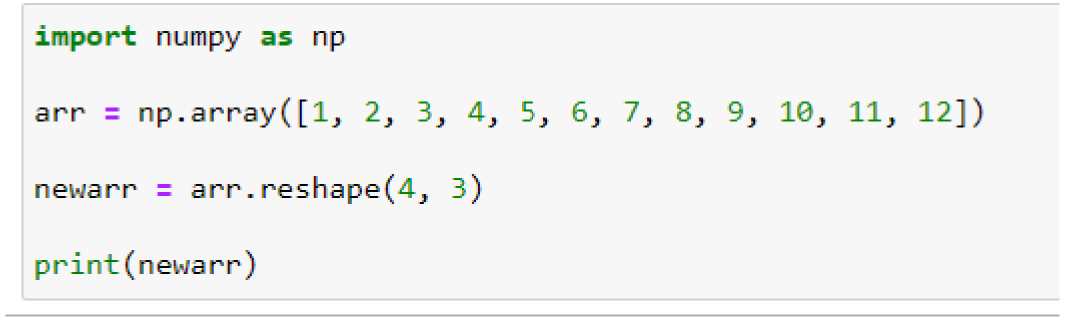
Reshaping arrays

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 number of elements in each dimension
.

Reshape From 1-D to 2-D

Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements:

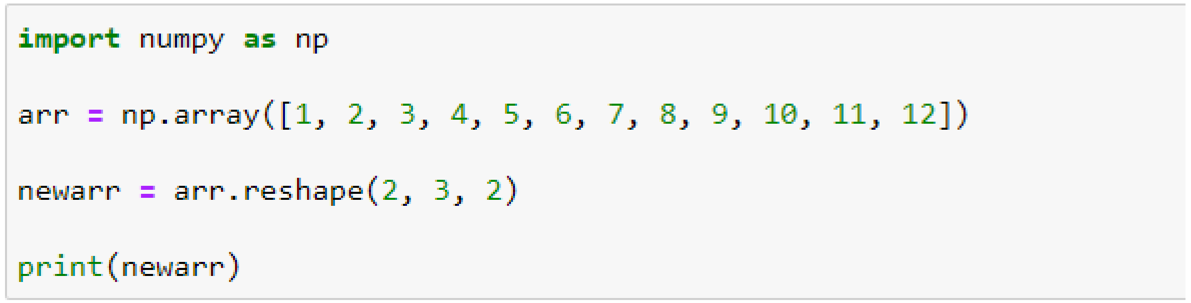


```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

Reshape From 1-D to 3-D

Convert the following 1-D array with 12 elements into a 3-D array.

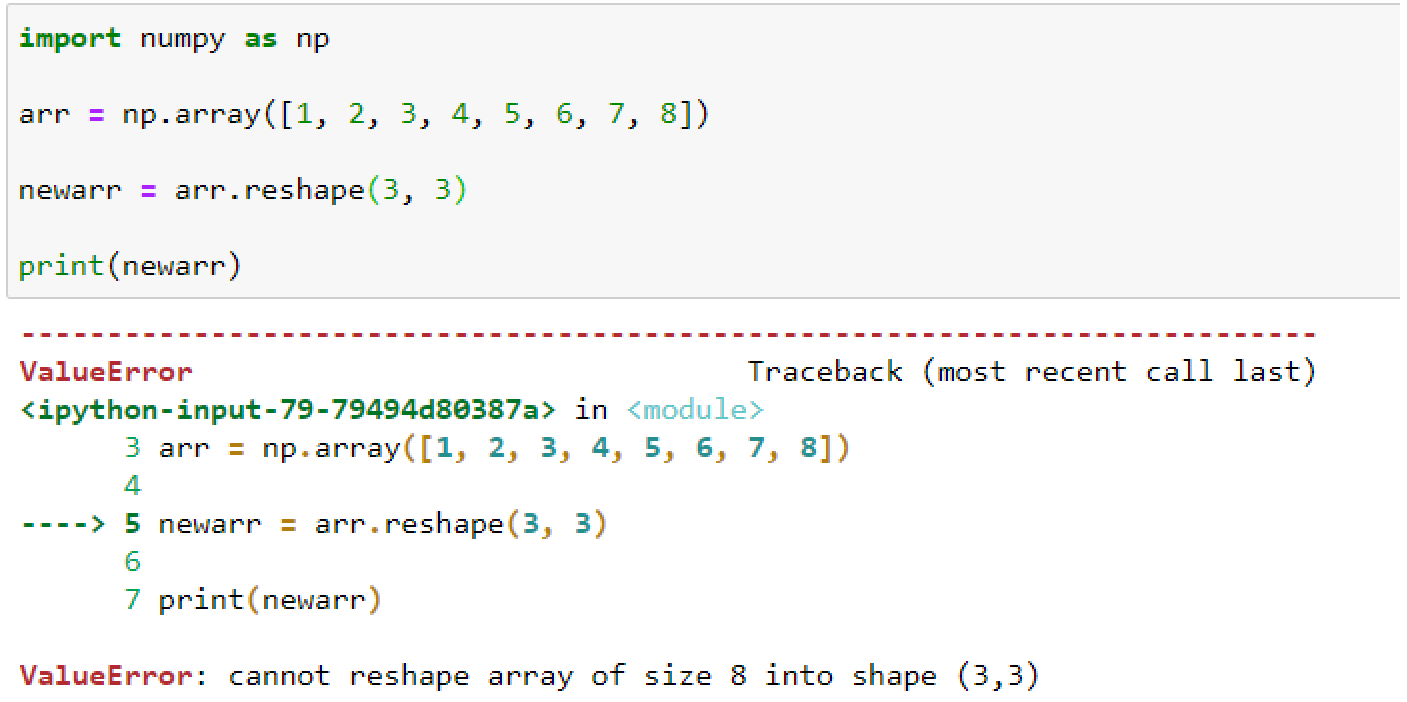
The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:



```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

Can We Reshape Into any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.
We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.



```
import numpy as np

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

newarr = arr.reshape(3, 3)

print(newarr)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-79-79494d80387a> in <module>
      3 arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
      4 
----> 5 newarr = arr.reshape(3, 3)
      6 
      7 print(newarr)

ValueError: cannot reshape array of size 8 into shape (3,3)
```

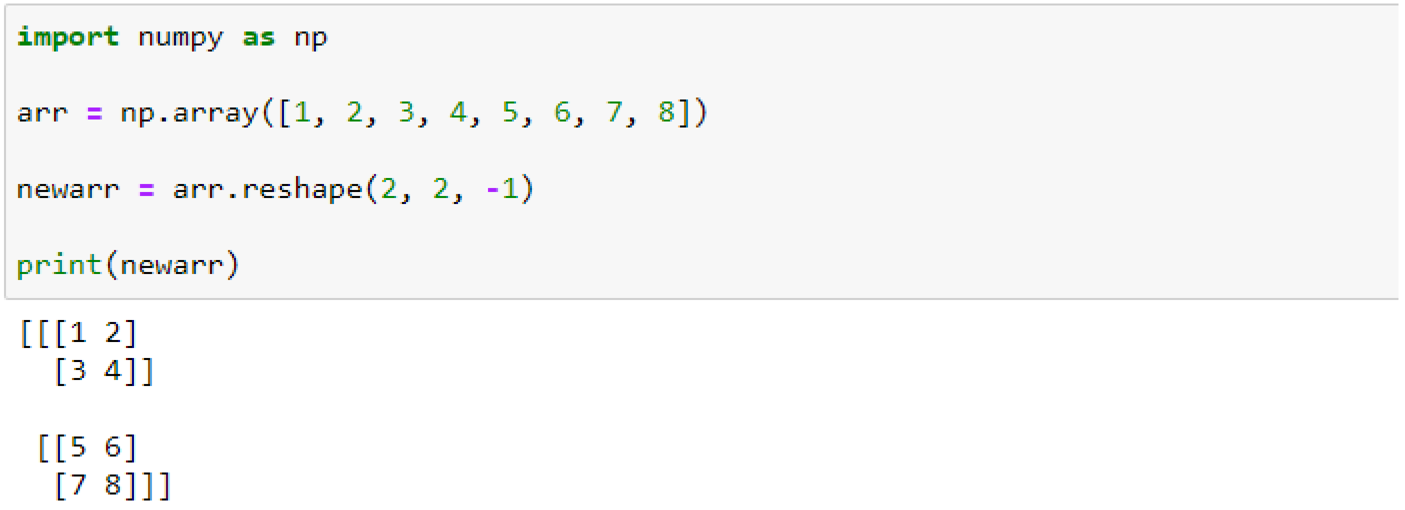
Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

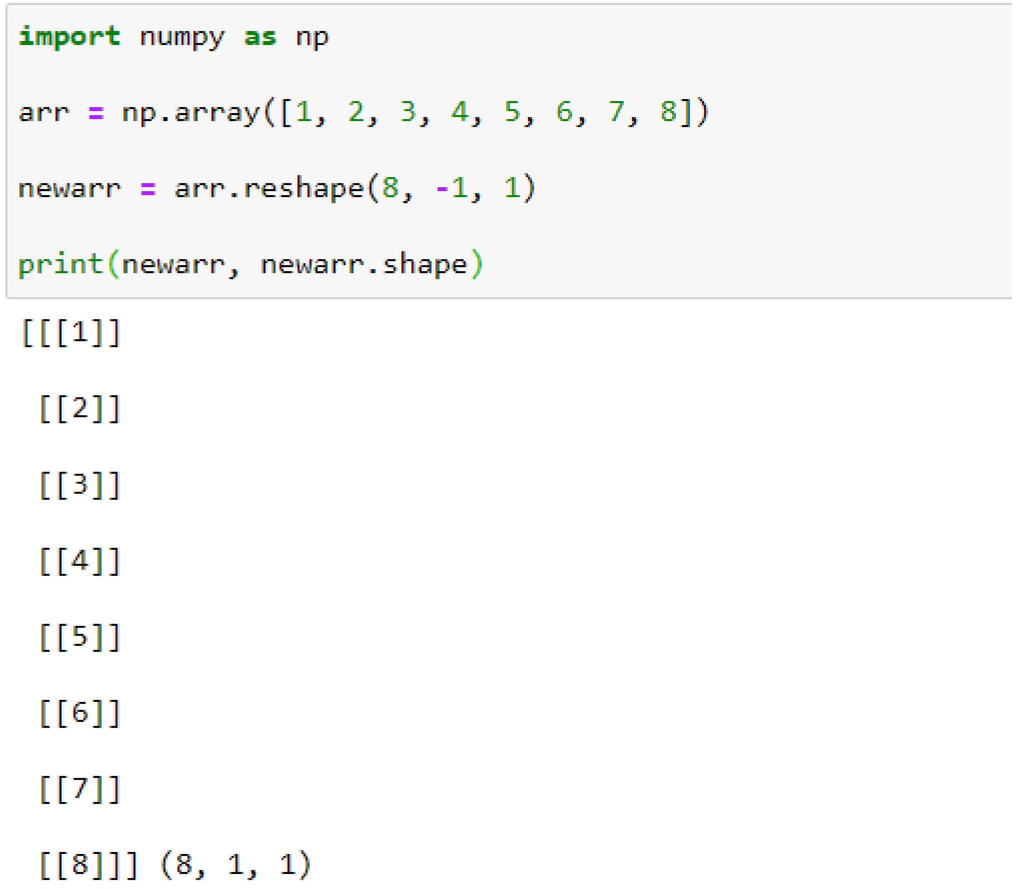
Pass -1 as the value, and NumPy will calculate this number for you.

Convert 1D array with 8 elements to 3D array with 2x2 elements:



```python
import numpy as np

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

newarr = arr.reshape(2, 2, -1)

print(newarr)
```

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

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

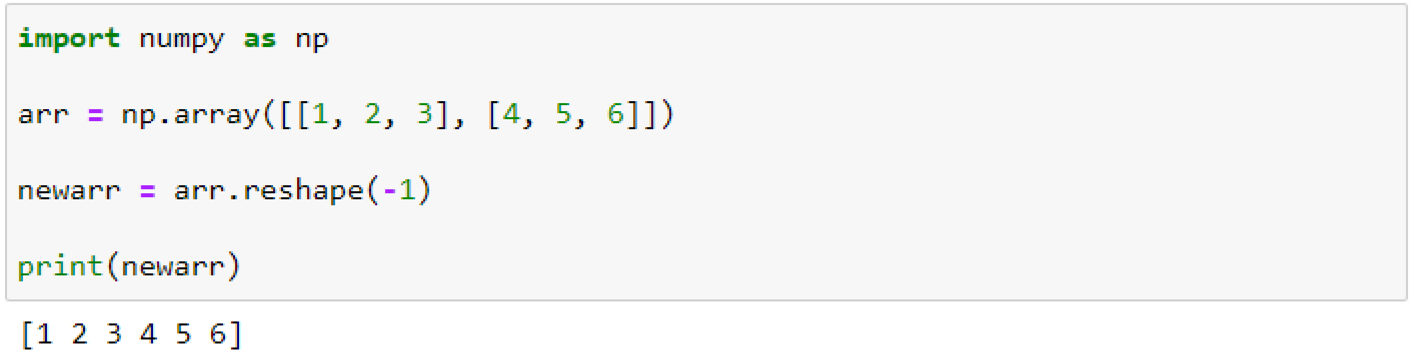


```python
import numpy as np

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

newarr = arr.reshape(8, -1, 1)

print(newarr, newarr.shape)
```

```
[[[1]]

 [[2]]

 [[3]]

 [[4]]

 [[5]]

 [[6]]

 [[7]]

 [[8]]] (8, 1, 1)
```

Flattening the arrays

Flattening array means converting a multidimensional array into a 1D array.
We can use reshape(-1) to do this



```python
import numpy as np

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

newarr = arr.reshape(-1)

print(newarr)
```

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

Intermediate NumPy

Session Objectives

At the end of this session, you will be able:

- ❑ Understand the Rules of broadcasting and its traps.
- ❑ Understand the Axis of rows and axis of columns of 2D array
- ❑ Aggregate data depending on different axis
- ❑ read a simple dataset with Numpy
- ❑ write to simple dataset with Nump

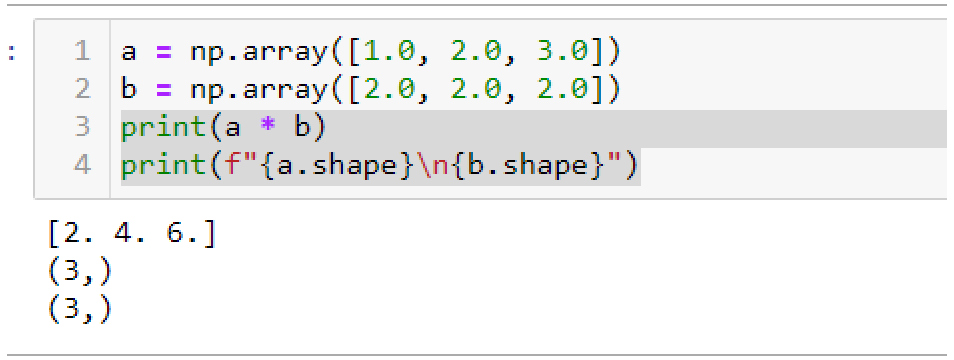
❑ Broadcasting:

https://numpy.org/doc/stable/user/theory.broadcasting.html#array-broadcasting-in-numpy

The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations. Subject to certain constraints, the smaller array is “broadcast” across the larger array so that they have compatible shapes. Broadcasting provides a means of vectorizing array operations so that looping occurs in C instead of Python. It does this without making needless copies of data and usually leads to efficient algorithm implementations. There are, however, cases where broadcasting is a bad idea because it leads to inefficient use of memory that slows computation.

NumPy operations are usually done on pairs of arrays on an element-by-element basis. In the simplest case, the two arrays must have exactly the same shape, as in the following example:

Example1:

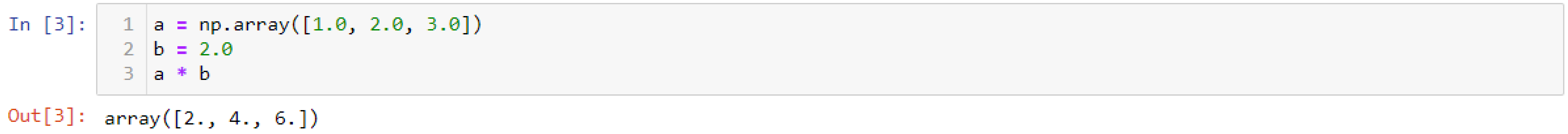


```
a = np.array([1.0, 2.0, 3.0])
b = np.array([2.0, 2.0, 2.0])
print(a * b)
print(f"{a.shape}\n{b.shape}")
```

```
[2. 4. 6.]
(3,)
(3,)
```

NumPy’s broadcasting rule relaxes this constraint when the arrays’ shapes meet certain constraints. The simplest broadcasting example occurs when an array and a scalar value are combined in an operation:

Example2:



```
In [3]: a = np.array([1.0, 2.0, 3.0])
        b = 2.0
        a * b

Out[3]: array([2., 4., 6.])
```

❑ General Broadcasting Rules

When operating on two arrays, NumPy compares their shapes element-wise. It starts with the trailing (i.e. rightmost) dimensions and works its way left. Two dimensions are compatible when they are equal, or one of them is 1

If these conditions are not met, a ValueError: operands could not be broadcast together exception is thrown, indicating that the arrays have incompatible shapes. The size of the resulting array is the size that is not 1 along each axis of the inputs.

Arrays do not need to have the same number of dimensions. For example, if you have a 256x256x3 array of RGB values, and you want to scale each color in the image by a different value, you can multiply the image by a one-dimensional array with 3 values. Lining up the sizes of the trailing axes of these arrays according to the broadcast rules, shows that they are compatible:

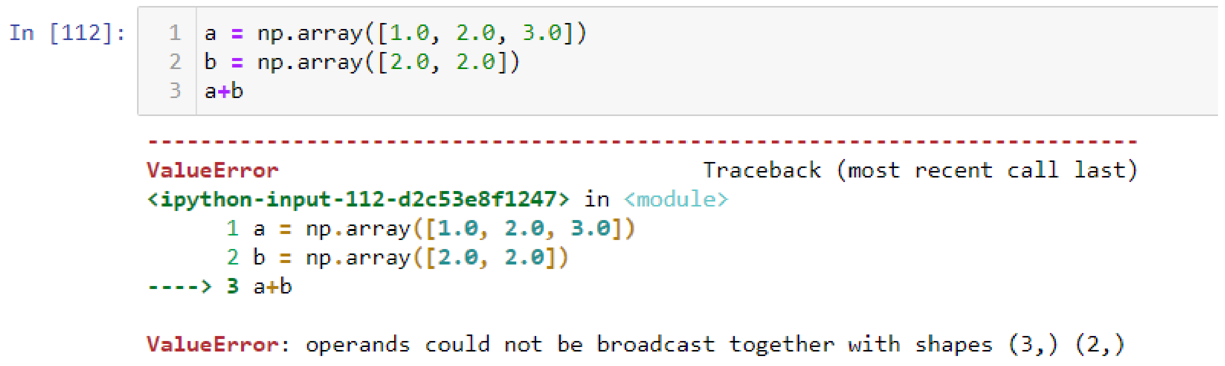
```
Image  (3d array): 256 x 256 x 3
Scale  (1d array):             3
Result (3d array): 256 x 256 x 3
```

When either of the dimensions compared is one, the other is used. In other words, dimensions with size 1 are stretched or “copied” to match the other.

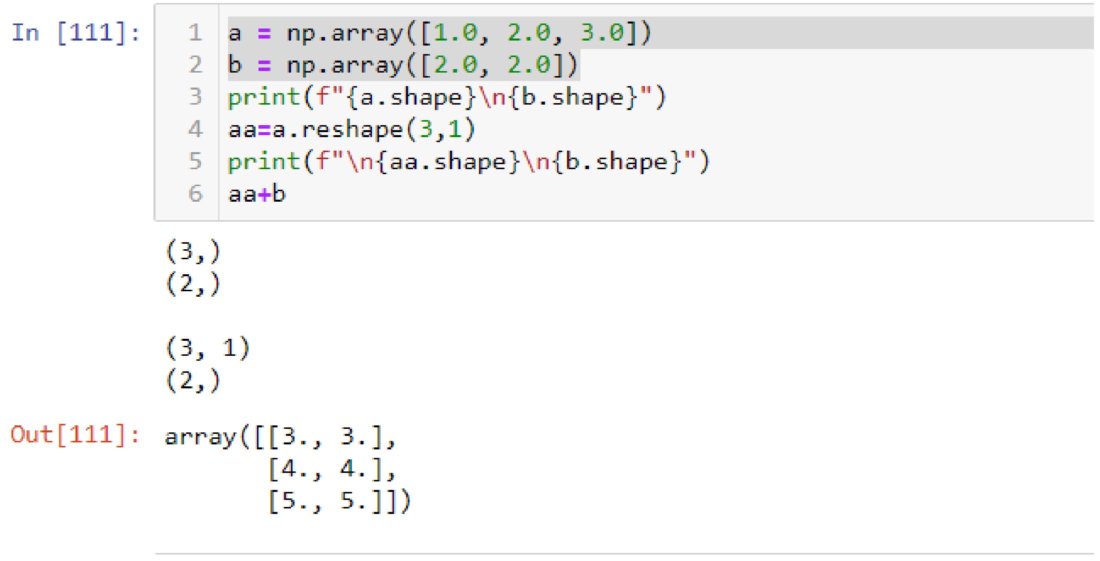
In the following example, both the A and B arrays have axes with length one that are expanded to a larger size during the broadcast operation:

```
A      (4d array):  8 x 1 x 6 x 1
B      (3d array):      7 x 1 x 5
Result (4d array):  8 x 7 x 6 x 5
```

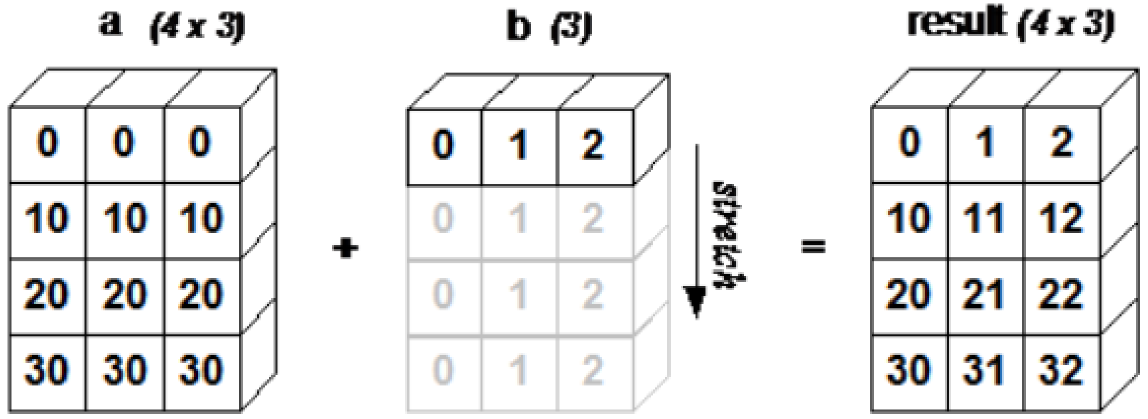
Example3:

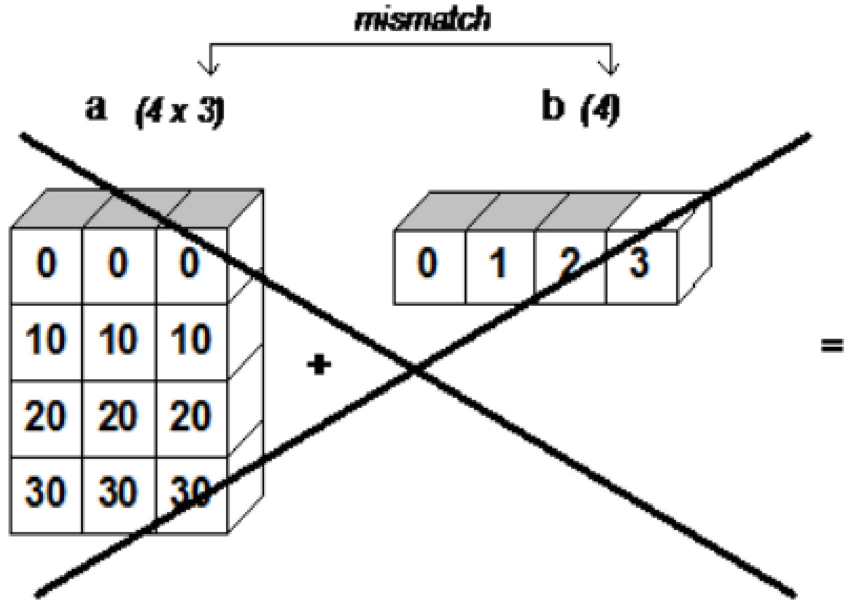


```
In [112]:  a = np.array([1.0, 2.0, 3.0])
           b = np.array([2.0, 2.0])
           a+b

---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-112-d2c53e8f1247> in <module>
      1 a = np.array([1.0, 2.0, 3.0])
      2 b = np.array([2.0, 2.0])
----> 3 a+b

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



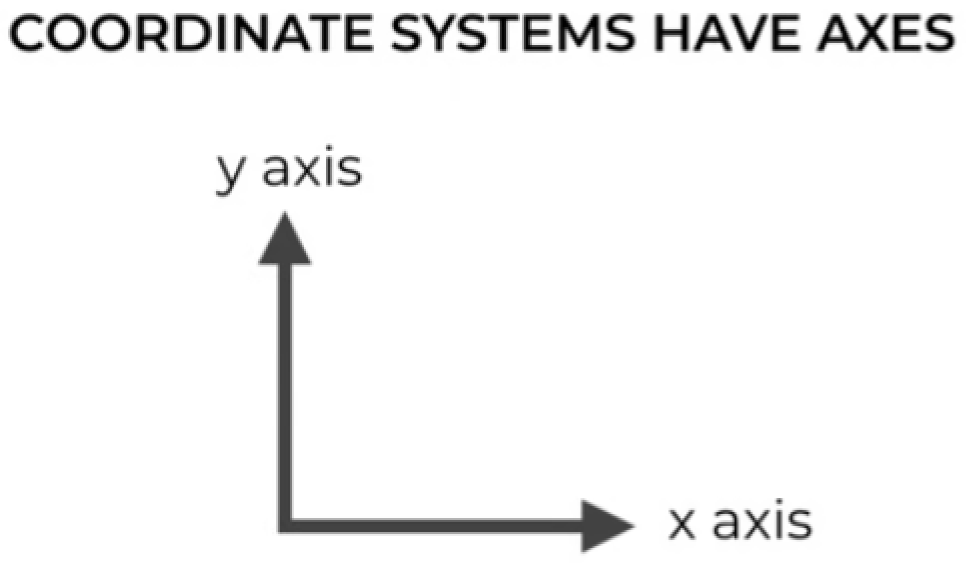
```
In [111]:  a = np.array([1.0, 2.0, 3.0])
           b = np.array([2.0, 2.0])
           print(f"{a.shape}\n{b.shape}")
           aa=a.reshape(3,1)
           print(f"\n{aa.shape}\n{b.shape}")
           aa+b

           (3,)
           (2,)

           (3, 1)
           (2,)

Out[111]:  array([[3., 3.],
                  [4., 4.],
                  [5., 5.]])
```

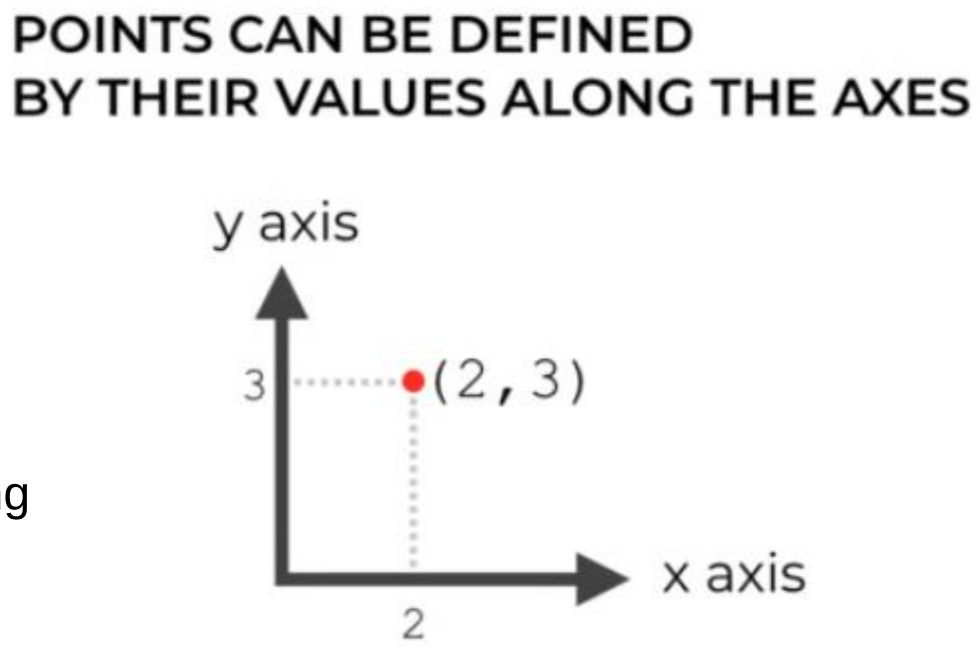

a (4 x 3)
b (3)
result (4 x 3)
stretch
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2
=
0 1 2
10 11 12
20 21 22
30 31 32


mismatch
a (4 x 3)
b (4)
0 0 0
10 10 10
20 20 20
30 30 30
+
0 1 2 3
=

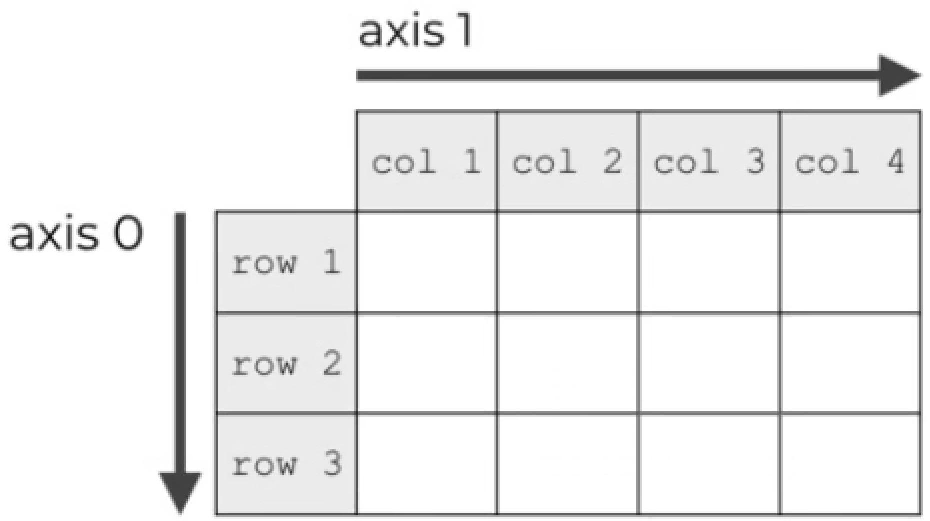
❑ Axis of rows and axis of columns of 2D array and Aggregate

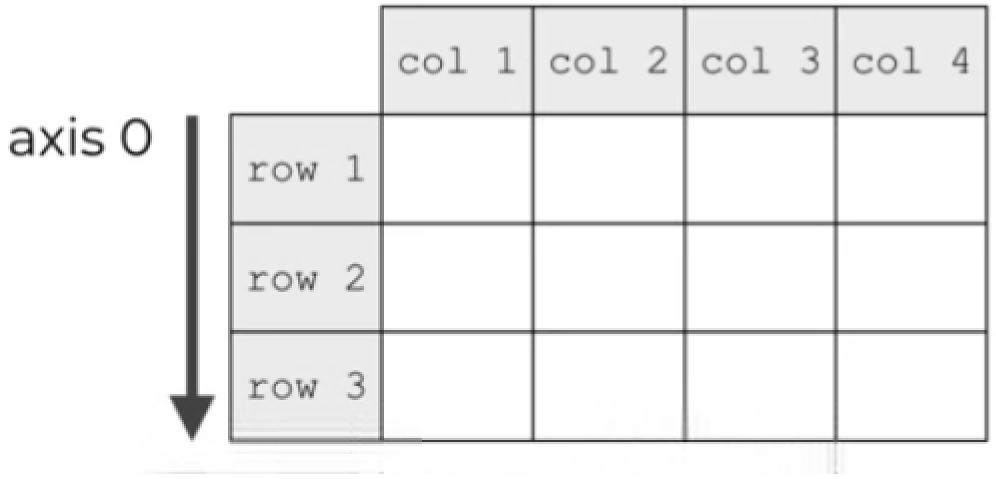
NUMPY AXES ARE LIKE AXES IN A COORDINATE SYSTEM





So if we have a point at position (2, 3), we're basically saying that it lies 2 units along the x axis and 3 units along the y axis.



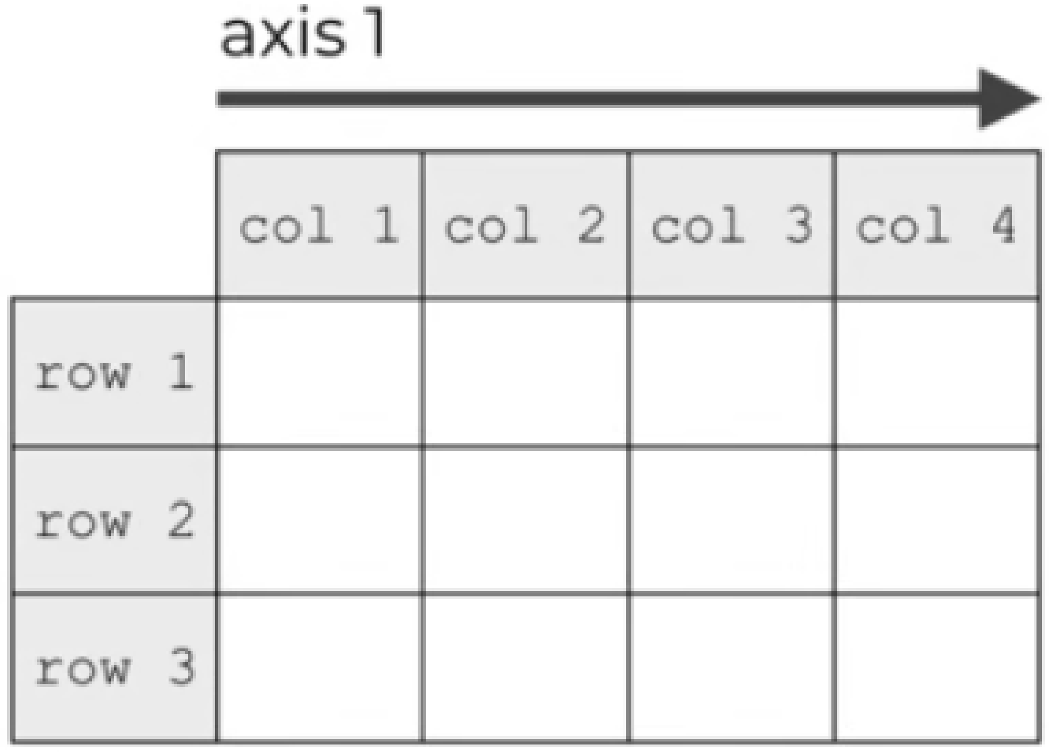


AXIS 0 IS THE DIRECTION ALONG THE ROWS
In a NumPy array, axis 0 is the “first” axis.
Assuming that we’re talking about multi-dimensional arrays, axis 0 is the axis that runs downward down the rows.

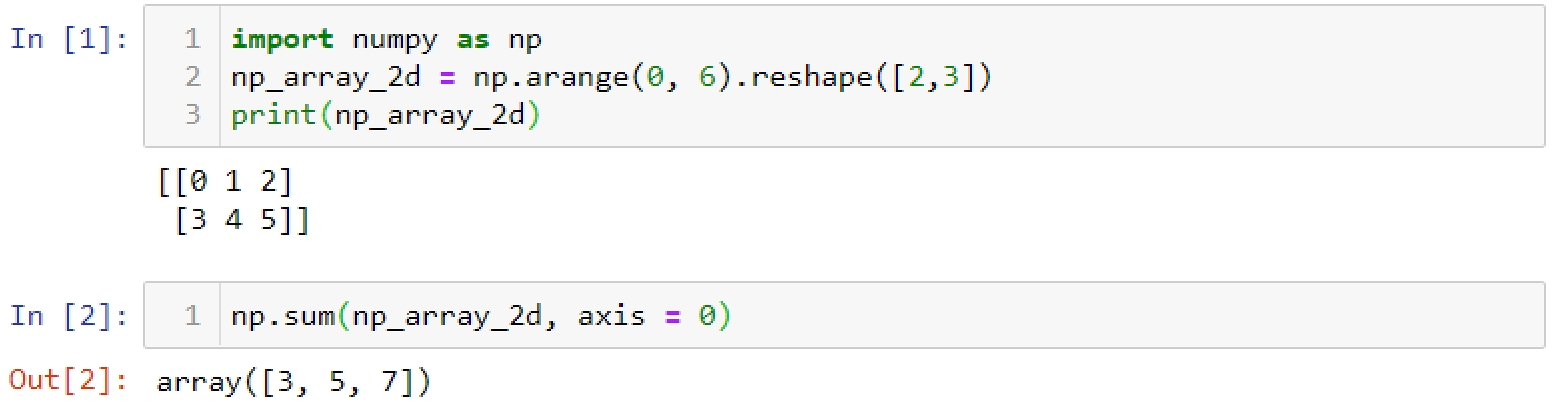
AXIS 1 IS THE DIRECTION ALONG THE COLUMNS

In a multi-dimensional NumPy array, axis 1 is the second axis.

When we're talking about 2-d and multi-dimensional arrays, axis 1 is the axis that runs horizontally across the columns.



EXAMPLES OF HOW NUMPY AXES ARE USED



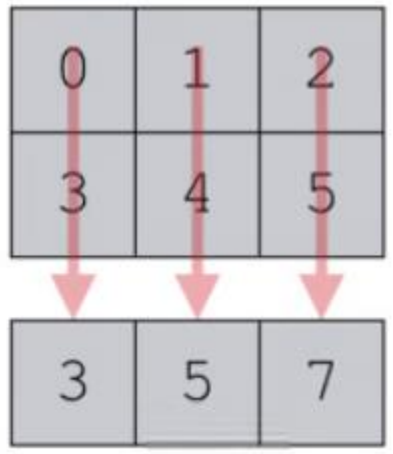
In [1]:
```python
import numpy as np
np_array_2d = np.arange(0, 6).reshape([2,3])
print(np_array_2d)
```

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

In [2]:
```python
np.sum(np_array_2d, axis = 0)
```

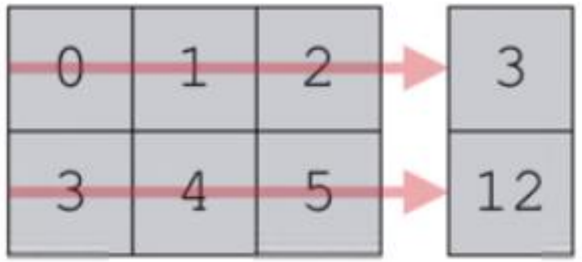
Out[2]:
```
array([3, 5, 7])
```

WHEN WE SET `axis = 0,np.sum()` COLLAPSES THE ROWS AND CALCULATES THE SUM



Why? Doesn’t axis 0 refer to the rows?

WHEN WE SET `axis = 1,np.sum()` COLLAPSES THE COLUMNS AND CALCULATES THE SUM



EXAMPLES 2

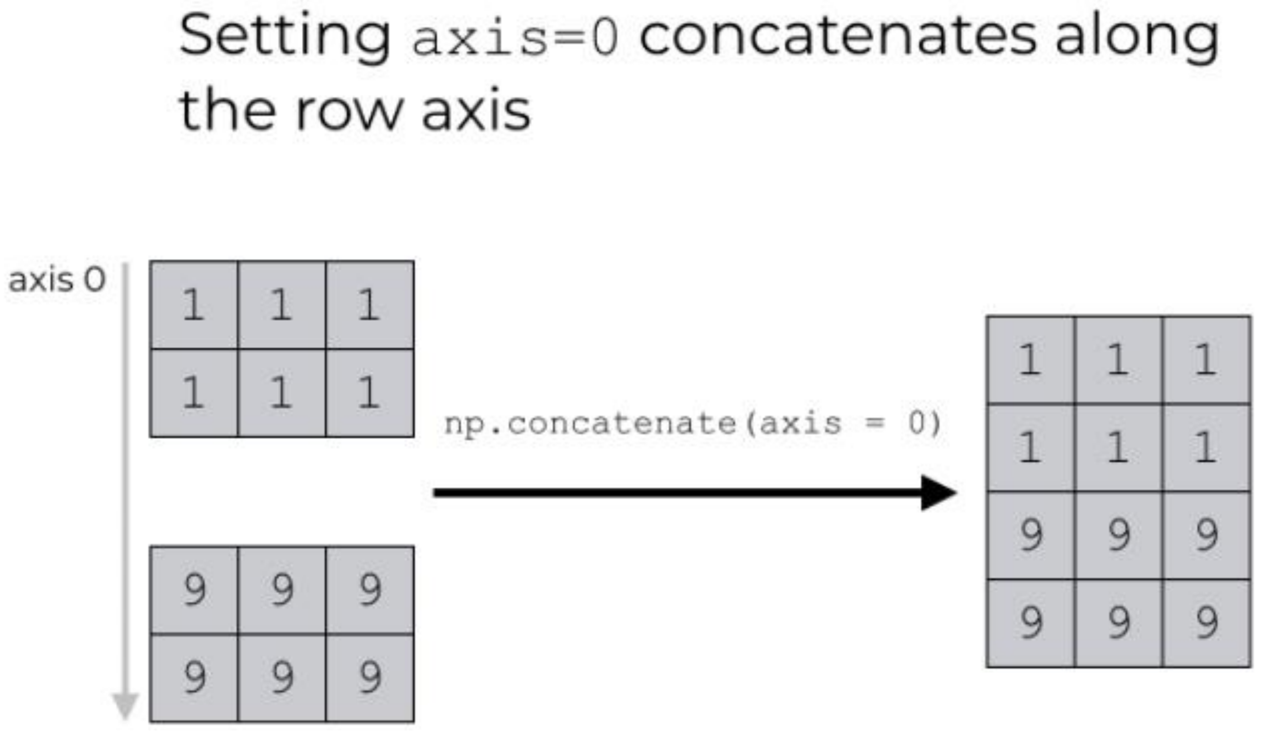
```
In [3]: print(np_array_2d)

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

In [4]: np.sum(np_array_2d, axis = 1)

Out[4]: array([ 3, 12])
```

Numpy Concatenation

```
In [8]:  np_array_1s = np.array([[1,1,1],[1,1,1]])
         np_array_9s = np.array([[9,9,9],[9,9,9]])
```

Numpy Concatenation on Zero Axis

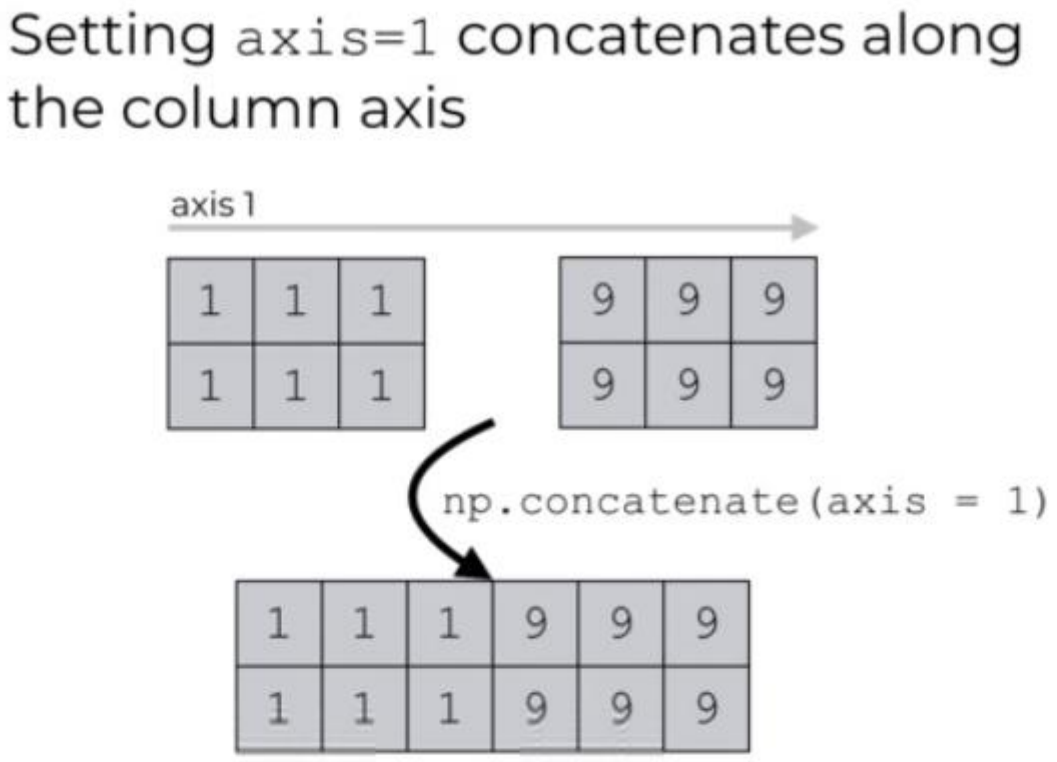
```
In [11]:  np.concatenate([np_array_1s, np_array_9s], axis = 0)

Out[11]: array([[1, 1, 1],
                [1, 1, 1],
                [9, 9, 9],
                [9, 9, 9]])
```



Numpy Concatenation on One Axis

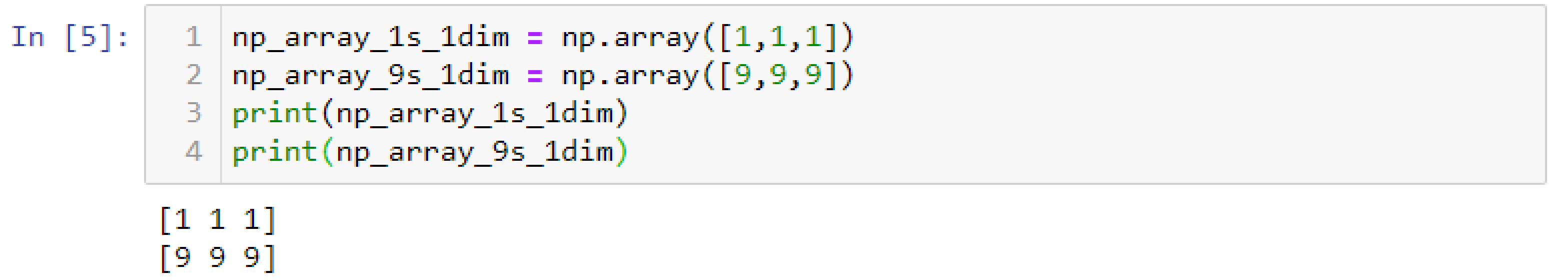
```
In [13]: np.concatenate([np_array_1s, np_array_9s], axis = 1)
```

```
Out[13]: array([[1, 1, 1, 9, 9, 9],
                [1, 1, 1, 9, 9, 9]])
```



WARNING: 1-DIMENSIONAL ARRAYS WORK DIFFERENTLY

The fact that 1-d arrays have only one axis (axis=0).

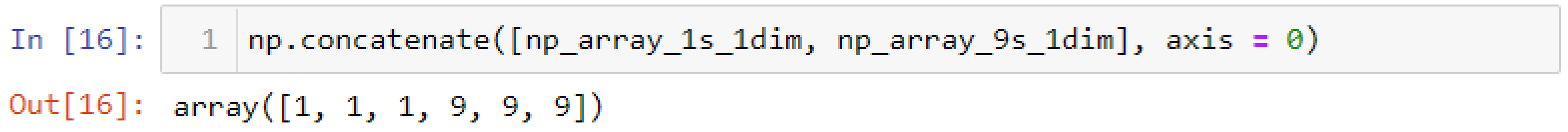


```
In [5]:
np_array_1s_1dim = np.array([1,1,1])
np_array_9s_1dim = np.array([9,9,9])
print(np_array_1s_1dim)
print(np_array_9s_1dim)

[1 1 1]
[9 9 9]
```

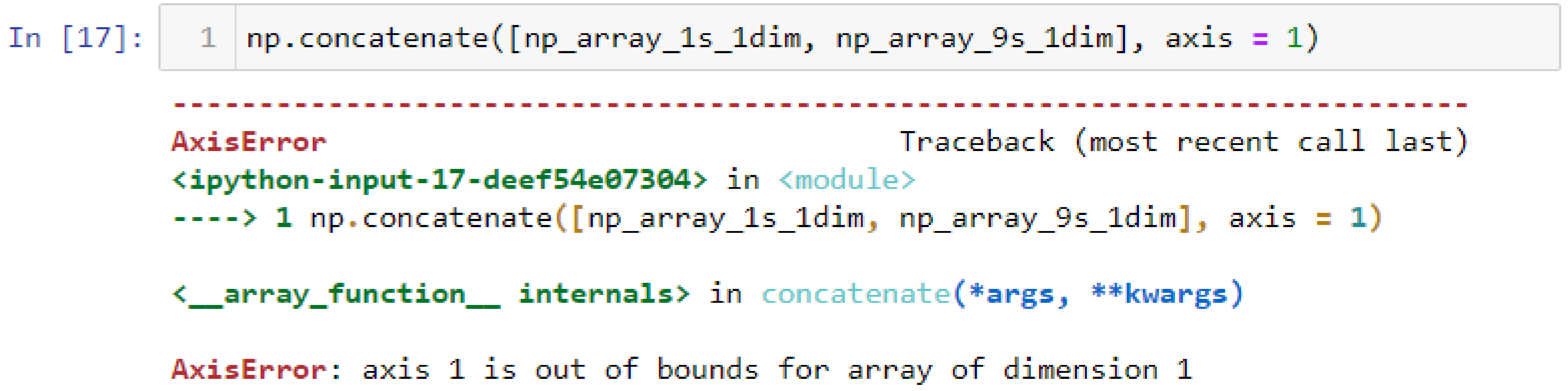
This is different from how the function works on 2-dimensional arrays. If we use np.concatenate() with axis = 0 on 2-dimensional arrays, the arrays will be concatenated together vertically.

What’s going on here?

Recall what I just mentioned a few paragraphs ago: 1-dimensional NumPy arrays only have one axis. Axis 0.



```
In [16]: np.concatenate([np_array_1s_1dim, np_array_9s_1dim], axis = 0)

Out[16]: array([1, 1, 1, 9, 9, 9])
```



```
In [17]: np.concatenate([np_array_1s_1dim, np_array_9s_1dim], axis = 1)
```

```
---------------------------------------------------------------------------
AxisError                                 Traceback (most recent call last)
<ipython-input-17-deef54e07304> in <module>
----> 1 np.concatenate([np_array_1s_1dim, np_array_9s_1dim], axis = 1)

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

AxisError: axis 1 is out of bounds for array of dimension 1
```

❑ Reading and Writing Data Files

there are lots of ways for reading from file and writing to data files in numpy. We will discuss the different ways and corresponding functions in this chapter:

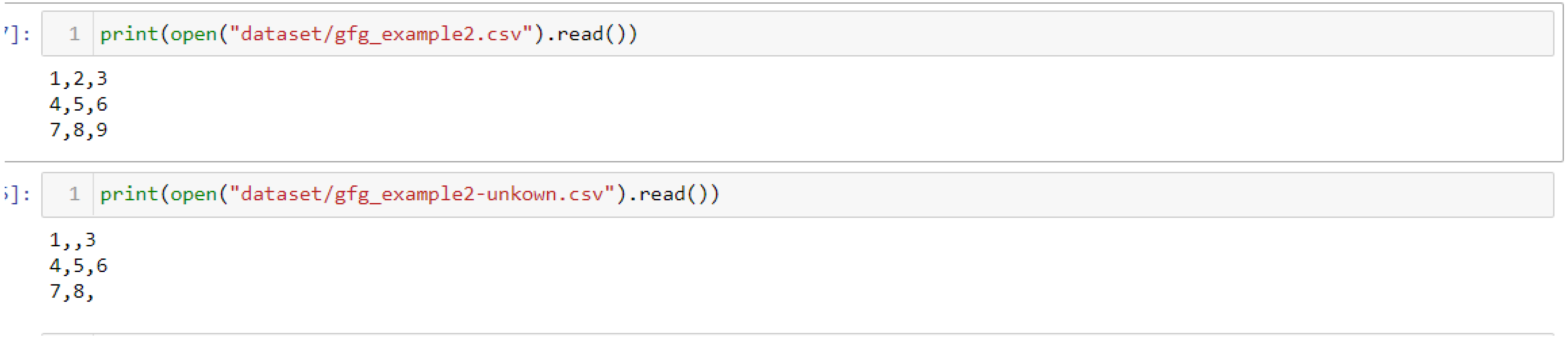
- genfromtxt
- savetxt
- loadtxt
- tofile
- fromfile
- save
- load

❑ Read simple dataset with Numpy

With missing values Use numpy.genfromtxt().

It return a masked array masking out

1. missing values (if usemask=True), or
2. fill in the missing value with the value specified in filling_values (default is np.nan for float, -1 for int).



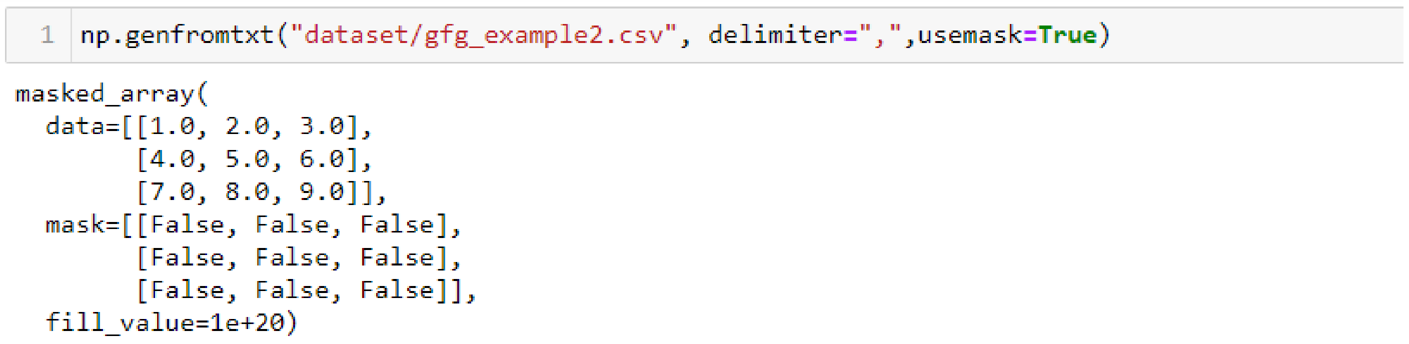
```
]: print(open("dataset/gfg_example2.csv").read())
```

```
1,2,3
4,5,6
7,8,9
```

```
]: print(open("dataset/gfg_example2-unkown.csv").read())
```

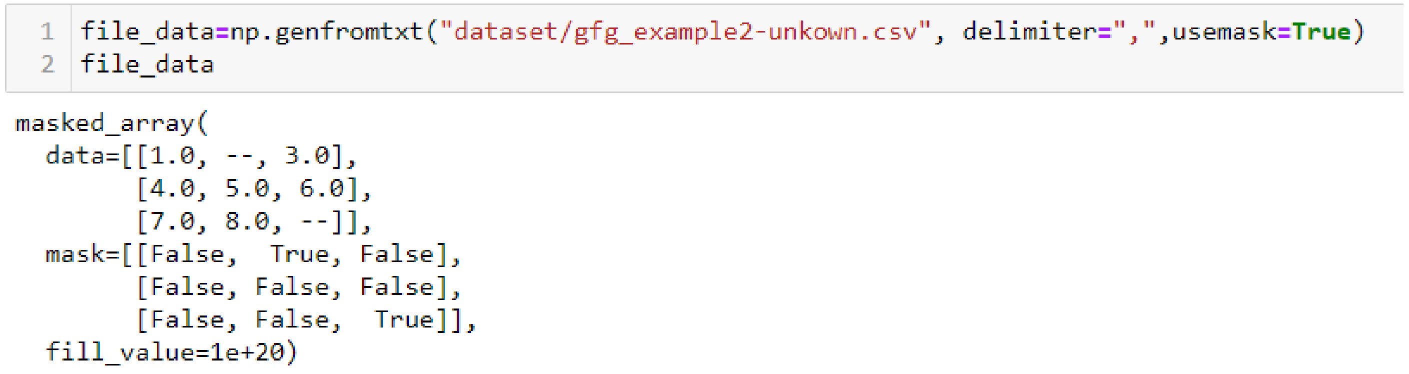
```
1,,3
4,5,6
7,8,
```

using usemask = True :



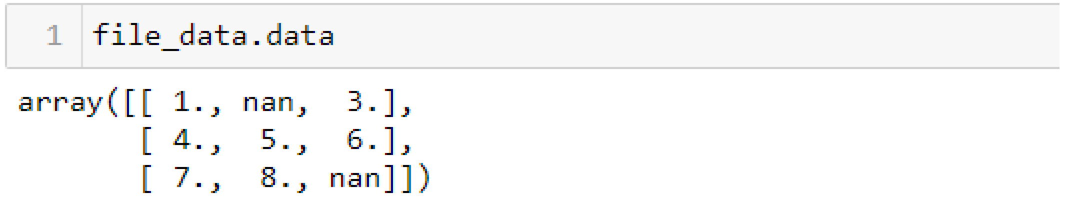
```
np.genfromtxt("dataset/gfg_example2.csv", delimiter=",",usemask=True)
```

```
masked_array(
  data=[[1.0, 2.0, 3.0],
        [4.0, 5.0, 6.0],
        [7.0, 8.0, 9.0]],
  mask=[[False, False, False],
        [False, False, False],
        [False, False, False]],
  fill_value=1e+20)
```



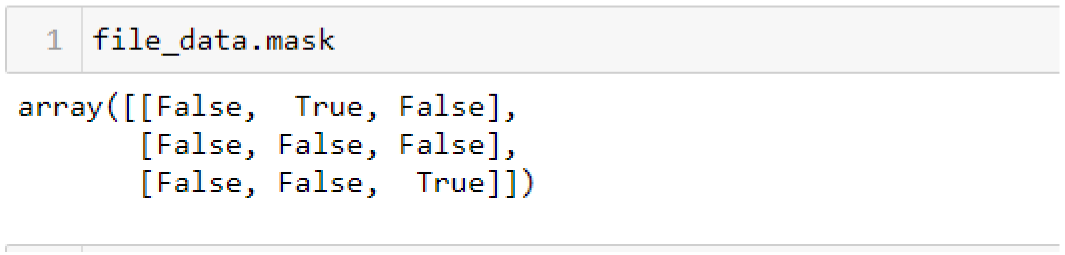
```
file_data=np.genfromtxt("dataset/gfg_example2-unkown.csv", delimiter=",",usemask=True)
file_data
```

```
masked_array(
  data=[[1.0, --, 3.0],
        [4.0, 5.0, 6.0],
        [7.0, 8.0, --]],
  mask=[[False,  True, False],
        [False, False, False],
        [False, False,  True]],
  fill_value=1e+20)
```



```
file_data.data
```

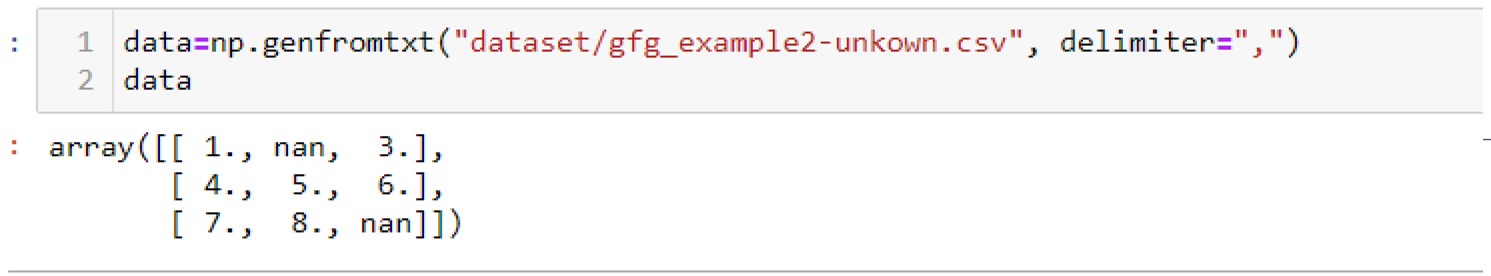
```
array([[ 1., nan,  3.],
       [ 4.,  5.,  6.],
       [ 7.,  8., nan]])
```



```
file_data.mask
```

```
array([[False,  True, False],
       [False, False, False],
       [False, False,  True]])
```

Without usemask:

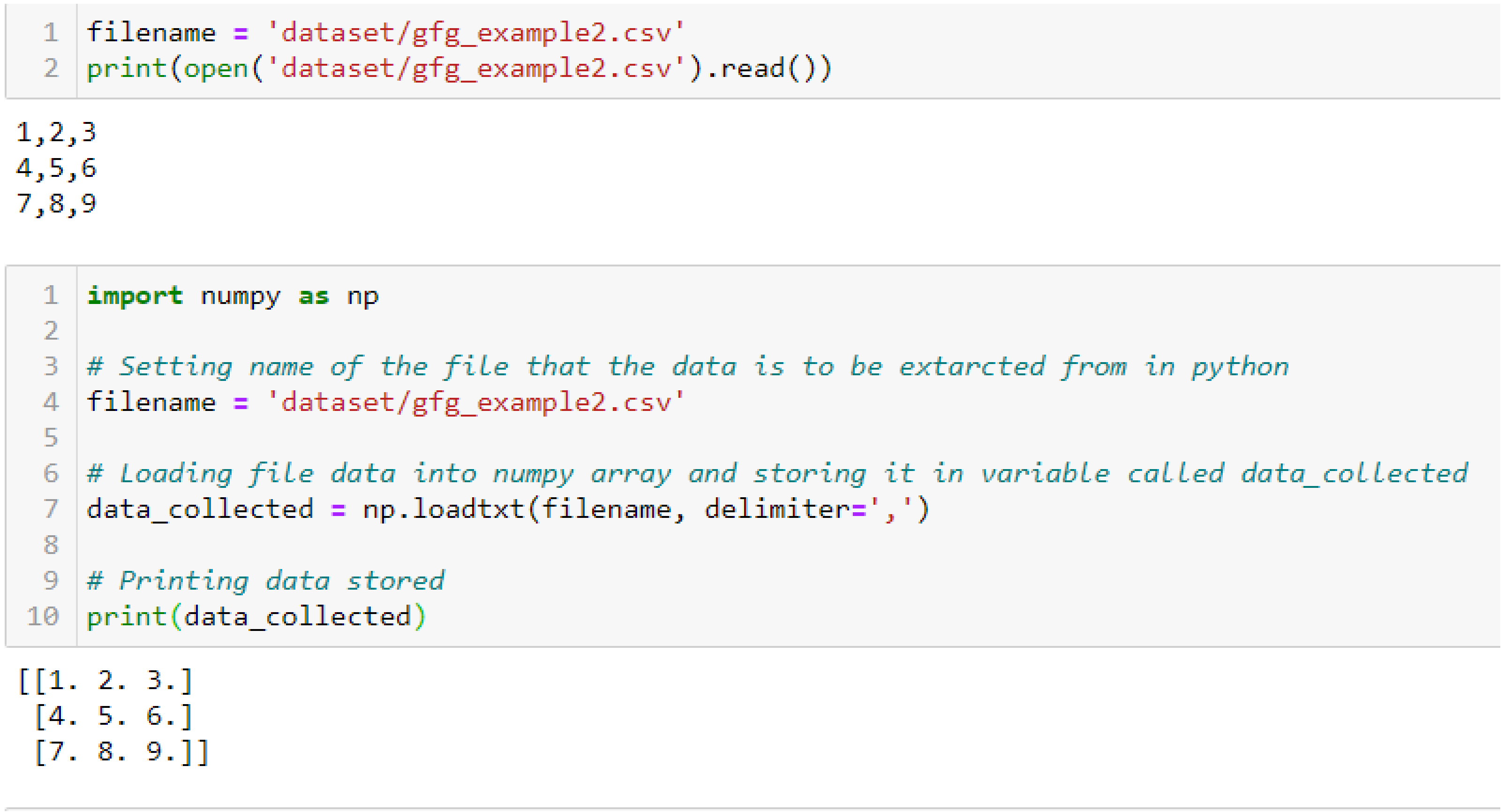


```
: data=np.genfromtxt("dataset/gfg_example2-unkown.csv", delimiter=",")
  data
```

```
: array([[ 1., nan,  3.],
         [ 4.,  5.,  6.],
         [ 7.,  8., nan]])
```

If data doesn’t contain missing values, you can use numpy.loadtxt(filepath,delimiter)

Example1:



```python
filename = 'dataset/gfg_example2.csv'
print(open('dataset/gfg_example2.csv').read())
```

```
1,2,3
4,5,6
7,8,9
```

```python
import numpy as np

# Setting name of the file that the data is to be extarcted from in python
filename = 'dataset/gfg_example2.csv'

# Loading file data into numpy array and storing it in variable called data_collected
data_collected = np.loadtxt(filename, delimiter=',')

# Printing data stored
print(data_collected)
```

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

Example2:

```
1  print(open("dataset/gfg_example2-unkown.csv").read())
```

```
1,,3
4,5,6
7,8,
```

```
 1  import numpy as np
 2  
 3  # Setting name of the file that the data is to be extarcted from in python
 4  filename = 'dataset/gfg_example2-unkown.csv'
 5  
 6  # Loading file data into numpy array and storing it in variable called data_collected
 7  data_collected = np.loadtxt(filename, delimiter=',')
 8  
 9  # Printing data stored
10  print(data_collected)
```

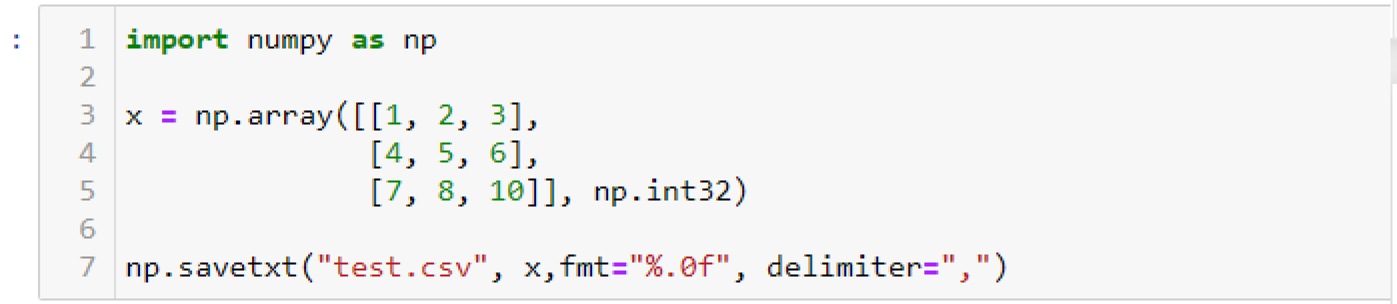
```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-64-e1b258fa3930> in <module>
      5 
      6 # Loading file data into numpy array and storing it in variable called data_collected
----> 7 data_collected = np.loadtxt(filename, delimiter=',')
      8 
      9 # Printing data stored
```

```
ValueError: could not convert string to float:
```

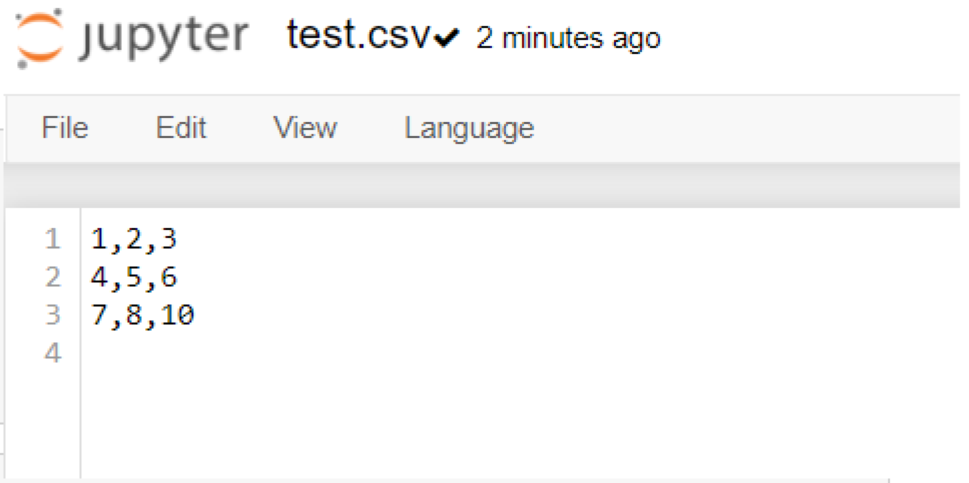
❑ Write simple dataset to file using Numpy

numpy.savetxt(filepath,data,delimiter,fmt)

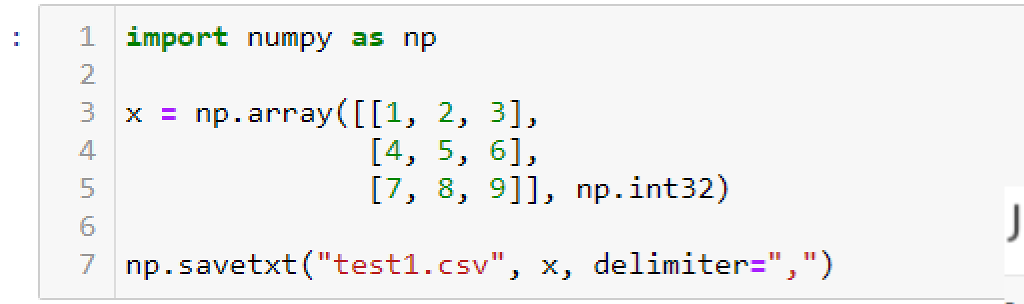
Example1:



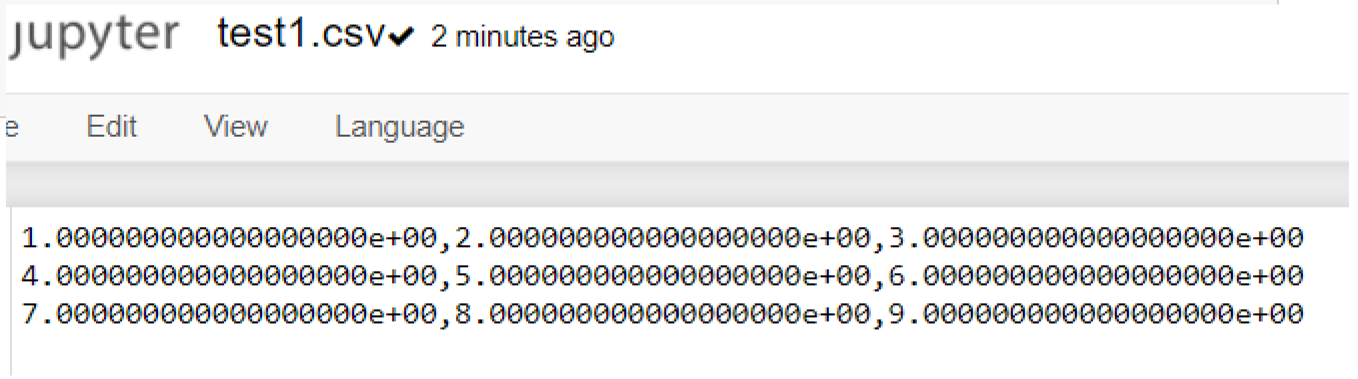
```python
import numpy as np

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 10]], np.int32)

np.savetxt("test.csv", x,fmt="%.0f", delimiter=",")
```



```
1,2,3
4,5,6
7,8,10
```

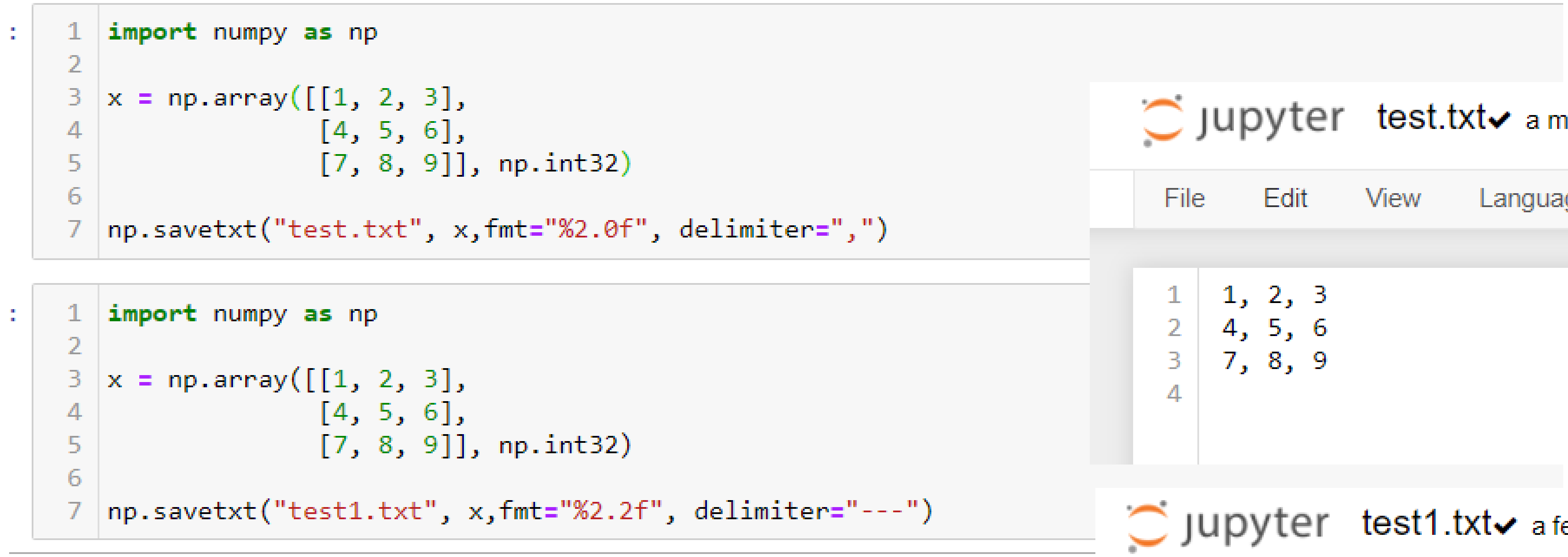


```python
import numpy as np

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], np.int32)

np.savetxt("test1.csv", x, delimiter=",")
```



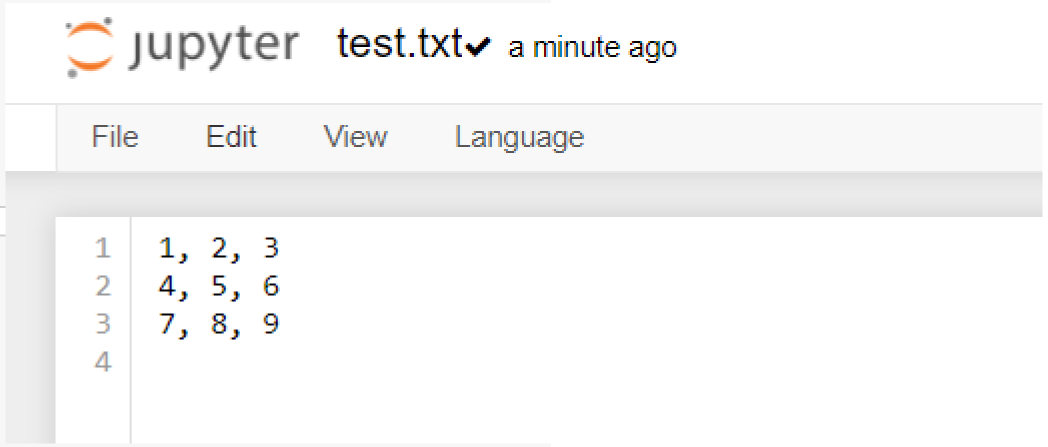
```
1.000000000000000000e+00,2.000000000000000000e+00,3.000000000000000000e+00
4.000000000000000000e+00,5.000000000000000000e+00,6.000000000000000000e+00
7.000000000000000000e+00,8.000000000000000000e+00,9.000000000000000000e+00
```

Example2:

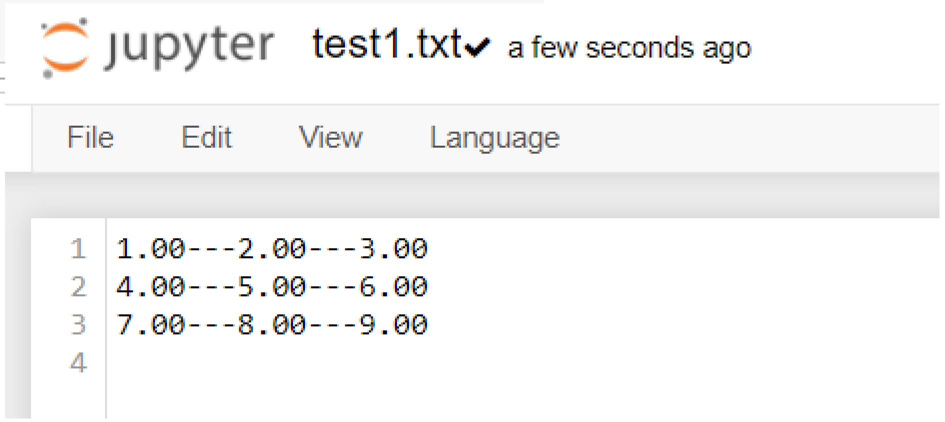


```
import numpy as np

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], np.int32)

np.savetxt("test.txt", x,fmt="%2.0f", delimiter=",")
```

```
import numpy as np

x = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]], np.int32)

np.savetxt("test1.txt", x,fmt="%2.2f", delimiter="---")
```



```
1, 2, 3
4, 5, 6
7, 8, 9
```



```
1.00---2.00---3.00
4.00---5.00---6.00
7.00---8.00---9.00
```

Lesson 1 presentation :

- Reading and Writing Data Files
- numpy.genfromtxt() vs numpy.loadtxt()
- numpy linear algebra methods
- Explain why Numpy data structures take up less space than pure python
- You can add any other numpy methods or functions (Extra points)