**NumPy**

**What is NumPy?**

NumPy is a python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. It is an open source project and you can use it freely. NumPy stands for Numerical Python.

**Where is NumPy used?**

Python NumPy arrays provide tools for integrating C, C++, etc. It is also useful in linear algebra, random number capability etc. NumPy array can also be used as an efficient multi-dimensional container for generic data. Now, let me tell you what exactly is a Python NumPy array.

**Why Use NumPy**

In Python we have lists that serve the purpose of arrays, but they are slow to process. NumPy aims to provide an array object that is up to 50x faster than traditional Python lists. The array object in NumPy is called **ndArray**, it provides a lot of supporting functions that make working with**ndarray** very easy.

Arrays are very frequently used in data science, where speed and resources are very important.

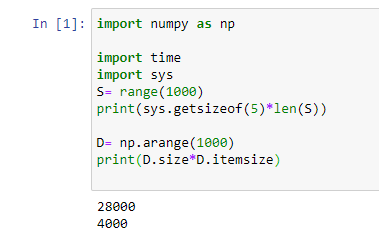
**Why NumPy is used in Python?**

We use python NumPy array instead of a list because of the below three reasons:

1. Less Memory
2. Fast
3. Convenient

The very first reason to choose python NumPy array is that it occupies less memory as compared to list. Then, it is pretty fast in terms of execution and at the same time, it is very convenient to work with NumPy. So these are the major advantages that Python NumPy array has over list. Don’t worry, I am going to prove the above points one by one practically in Jupyter Notebook.

Consider the below example:



The above output shows that the memory allocated by list (denoted by S) is 28000 whereas the memory allocated by the NumPy array is just 4000. From this, you can conclude that there is a major difference between the two and this makes Python NumPy array as the preferred choice over list.

**Why Numpy is Faster than Lists**

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently. This behavior is called locality of reference in computer science. This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

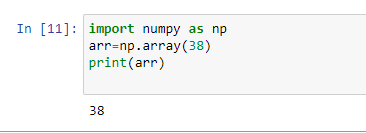
**Dimensions in Arrays**

A dimension in array is one level of array depth (nested arrays). **Nested Arrays** are arrays that have arrays as their elements.

1. **0-D Arrays:**

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array

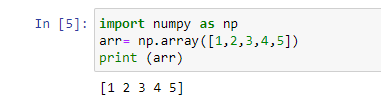
**Create a 0-D array with value 38**



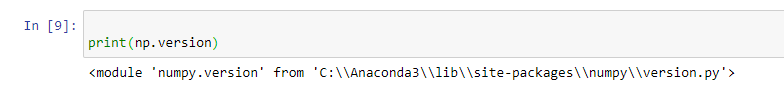
1. **1-D Arrays:**

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.

Create a 1-D array containing the values 1,2,3,4,5:



**Checking the NumPy Version:**



**type():** This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy. array type.

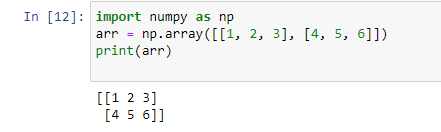
To create an  ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

1. **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.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

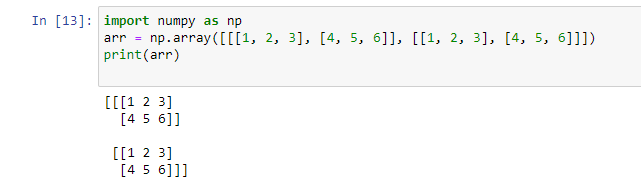
Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:



1. **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.

Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

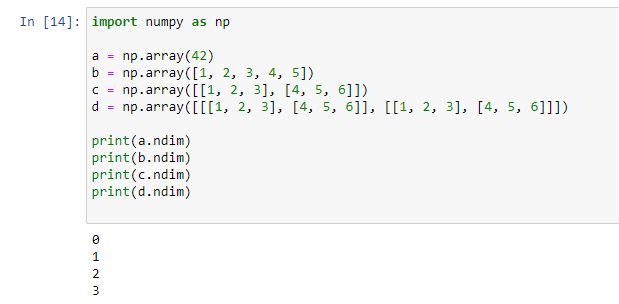


**Check Number of Dimensions?**

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example

Check how many dimensions the arrays have:

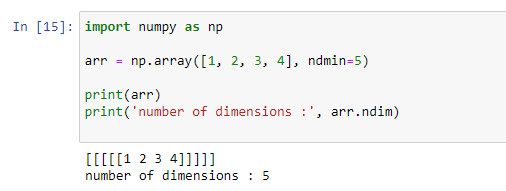


1. **Higher Dimensional Arrays**

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.

Create an array with 5 dimensions and verify that it has 5 dimensions:

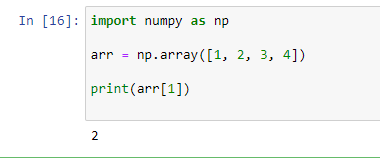


**NumPy Array Indexing**

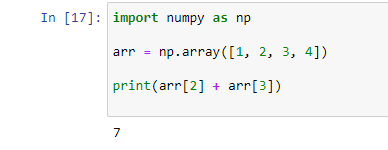
**Access the Array elements**

Array indexing is the same as accessing an array element. 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.

1. Get Second element From the Array:



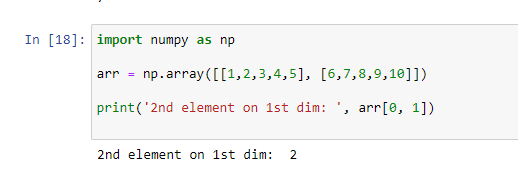
1. Get Third element and fourth elements from the array and add them



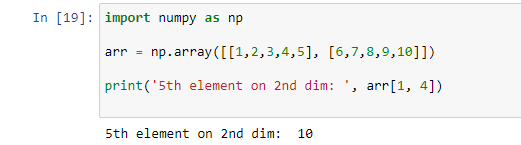
1. **Access 2-D Arrays**

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

1. Access the 2nd element on 1st dim:



1. Access the 5th element on 2nd dim:

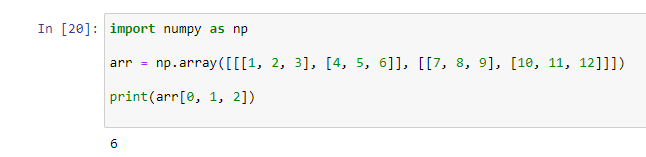


1. **Access 3-D Arrays**

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

**Example**

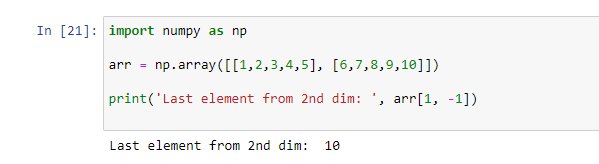
Access the third element of the second array of the first array:



1. **Negative Indexing**

Print the last element from the 2nd dim:

Use negative indexing to access an array from the end.

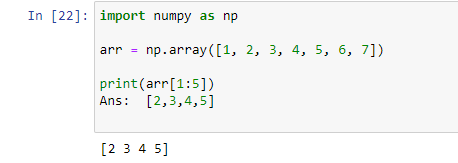


**NumPy Array Slicing**

**Slicing arrays**

* 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

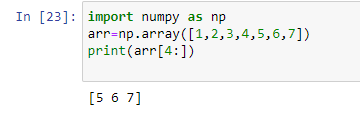
Slice elements from index 1 to index 5 from the following array:



Note: The result includes the start index, but excludes the end index.

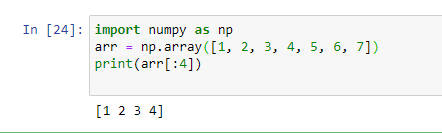
**Example :1**

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



**Example :2**

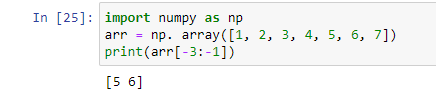
Slice elements from the beginning to index 4 (not included):



**Negative Slicing**

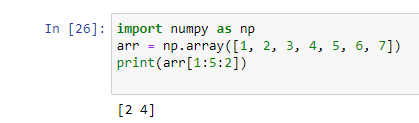
Use the minus operator to refer to an index from the end:

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

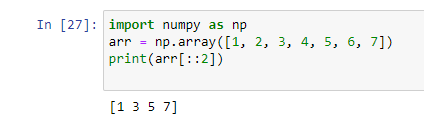


**STEP:-**

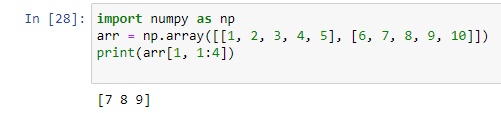
Use the step value to determine the step of the slicing: Return every other element from index 1 to index 5:



Return every other element from the entire array:

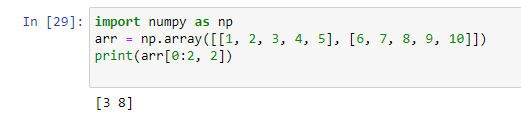


**Slicing 2-D Arrays**

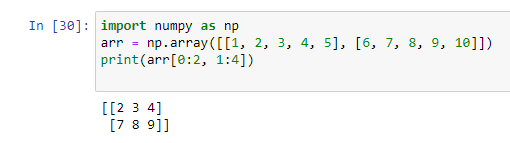
1. From the second element, slice elements from index 1 to index 4 (not included)  
     
   

**Note:** Remember that second element has index 1.

1. From both elements, return index 2:

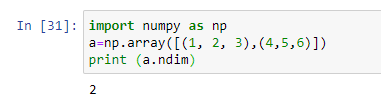


1. From both elements, slice index 1 to index 4 (not included), this will return a 2-D array:



**Python Numpy Operations**

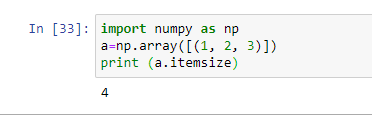
**ndim** : You can find the dimension of the array, whether it is a two-dimensional array or a single dimensional array. So, let us see this practically how we can find the dimensions. In the below code, with the help of ‘ndim’ function, I can find whether the array is of single dimension or multi dimension.



Since the output is 2, it is a two-dimensional array (multi dimension).

**Item Size:**

You can calculate the byte size of each element. In the below code, I have defined a single dimensional array and with the help of ‘itemsize’ function, we can find the size of each element.



So every element occupies 4 byte in the above numpy array.

**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.

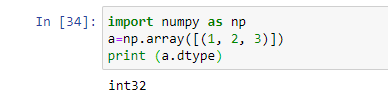
Below is a list of all data types in NumPy and the characters used to represent them.

* i - integer
* b - Boolean
* u - unsigned integer
* f - float
* c - complex float
* m - time delta
* M - datetime
* - object
* S - string
* U - unicode string
* V - fixed chunk of memory for other type (void)

**Checking the Data Type of an Array**

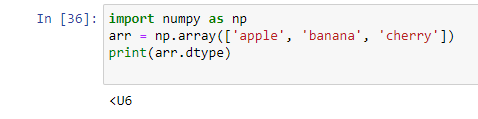
**dtype**

You can find the data type of the elements that are stored in an array. So, if you want to know the data type of a particular element, you can use **dtype** function which will print the datatype along with the size. In the below code, I have defined an array where I have used the same function.

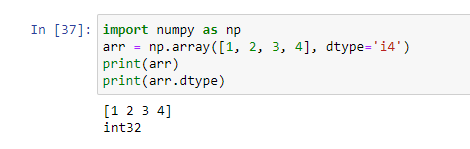


As you can see, the data type of the array is integer 32 bits. Similarly, you can find the size and shape of the array using ‘size’ and ‘shape’ function respectively.

1. Get the data type of an array containing strings:

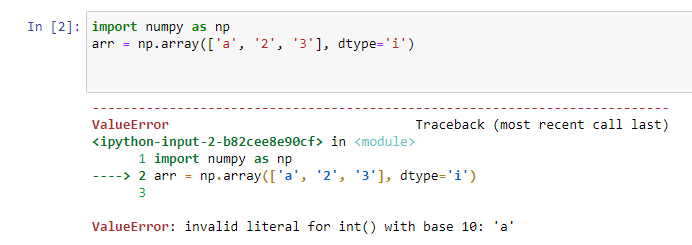


1. Create an array with data type 4 bytes integer:



If a type is given in which elements can't be casted then NumPy will raise a Value Error.

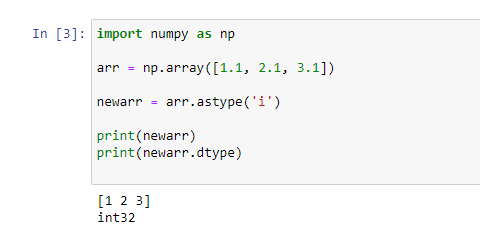
**ValueError:** In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

1. A non-integer string like 'a' cannot be converted to integer (will raise an error)  
   
2. Converting Data Type on Existing Arrays

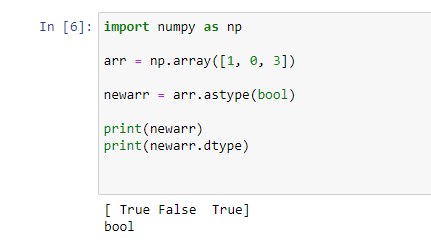
The best way to change the data type of an existing array, is to make a copy of the array with the**astype()** method. The **astype()** function creates a copy of the array, and allows you to specify the data type as a parameter.

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.

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



1. Change data type from integer to Boolean:



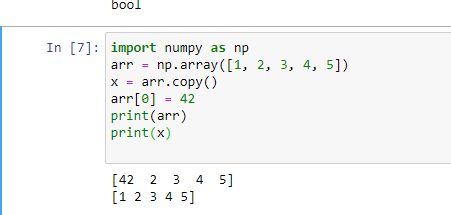
**Copy and View**

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

COPY:

The copy **owns**the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

Make a copy, change the original array, and display both arrays:

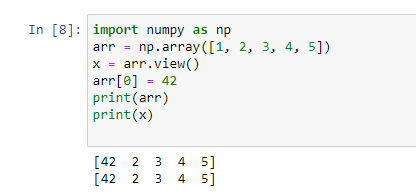


The copy SHOULD NOT be affected by the changes made to the original array.

VIEW:

The view **does not own** the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

Make a view, change the original array, and display both arrays:



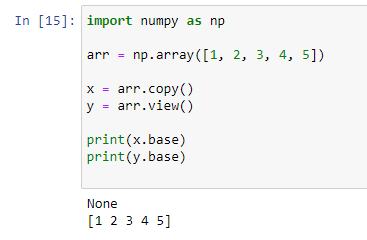
The view SHOULD be affected by the changes made to the original array.

**Check if Array Owns it's Data**

As mentioned above, copies owns the data, and views does not own the data, but how can we check this?

Every NumPy array has the attribute base that returns **None** if the array owns the data.

Otherwise, the **base** attribute refers to the original object.



Print the value of the base attribute to check if an array owns it's data or not:

The copy returns None.  
The view returns the original array.

**NumPy Array Shape**

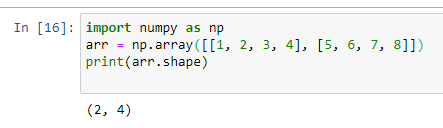
**Shape of an Array**

The shape of an array is the number of elements in each dimension.

Get the Shape of an Array

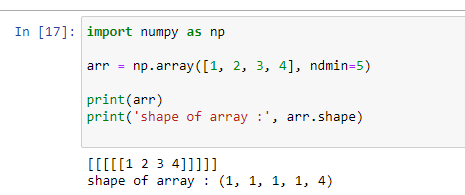
NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

Print the shape of a 2-D array:



The example above returns (2, 4), which means that the array has 2 dimensions, and each dimension has 4 elements.

Create an array with 5 dimensions using ndmin using a vector with values 1,2,3,4 and verify that last dimension has value 4:

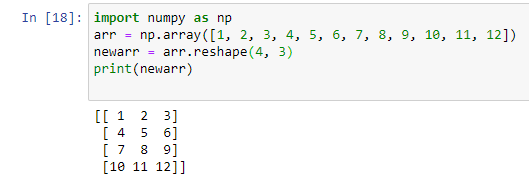


**Reshaping arrays**

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.

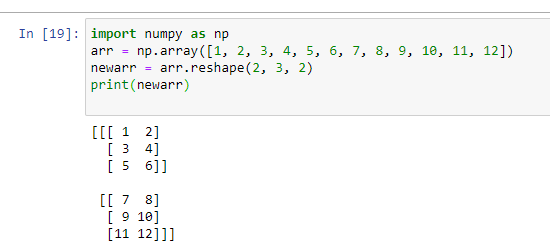
1. **Reshape From 1-D to 2-D**

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:



1. **Reshape From 1-D to 3-D**

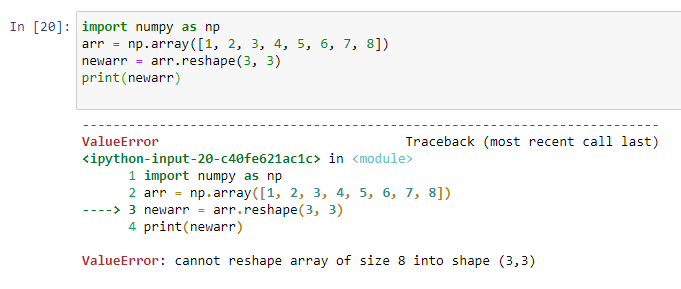
Convert the following 1-D array with 12 elements into a 3-D array. The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:



**Can We Reshape Into any Shape?**

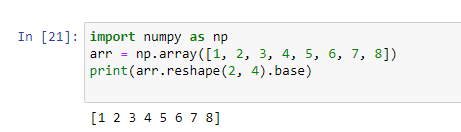
Yes, as long as the elements required for reshaping are equal in both shapes. We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

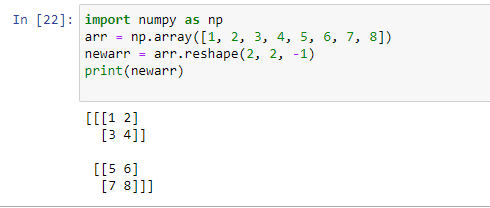


**Returns Copy or View?**

Check if the returned array is a copy or a view:



The example above returns the original array, so it is a view.

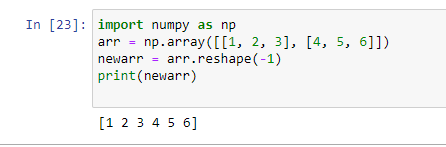


**Note:** We can not pass -1 to more than one dimension.

**Flattening the arrays**

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

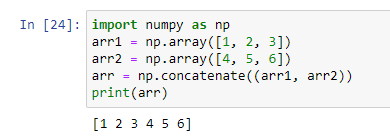
Convert the array into a 1D array:



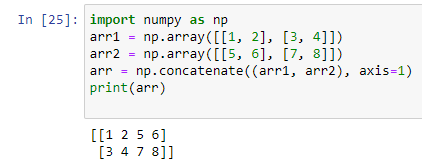
**Joining NumPy Arrays**

Joining means putting contents of two or more arrays in a single array. In SQL we join tables based on a key, whereas in NumPy we join arrays by axes. We pass a sequence of arrays that we want to join to the concatenate() function, along with the axis. If axis is not explicitly passed, it is taken as 0.

* **Join two arrays**



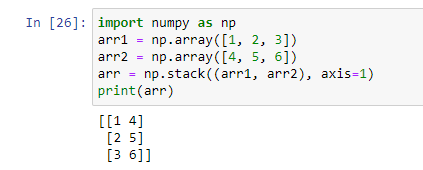
* **Join two 2-D arrays along rows (axis=1):**



* **Joining Arrays Using Stack Functions**

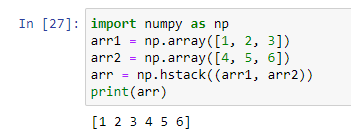
Stacking is same as concatenation; the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking. We pass a sequence of arrays that we want to join to the concatenate() method along with the axis. If axis is not explicitly passed it is taken as 0.



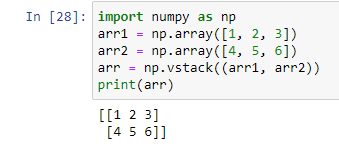
* **Stacking Along Rows**

NumPy provides a helper function: hstack() to stack along rows.



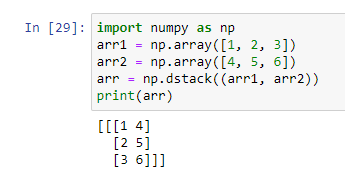
* **Stacking Along Columns**

NumPy provides a helper function: vstack()  to stack along columns.



* **Stacking Along Height (depth)**

NumPy provides a helper function: dstack() to stack along height, which is the same as depth.

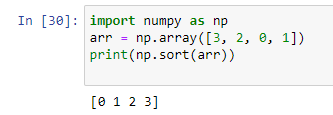


**Sorting Arrays**

Sorting means putting elements in a **ordered sequence**. Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called sort(), that will sort a specified array.

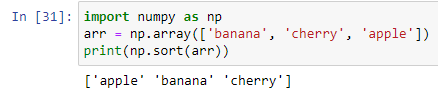
* **Sort the array:**



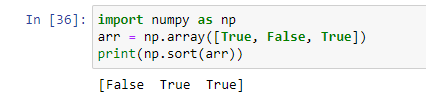
**Note:** This method returns a copy of the array, leaving the original array unchanged.

* **You can also sort arrays of strings, or any other data type:**

Sort the array alphabetically:

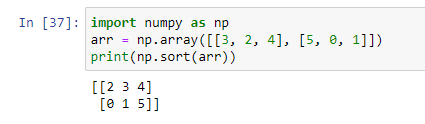


* **Sort a boolean array:**



* **Sorting a 2-D Array**

If you use the sort() method on a 2-D array, both arrays will be sorted:



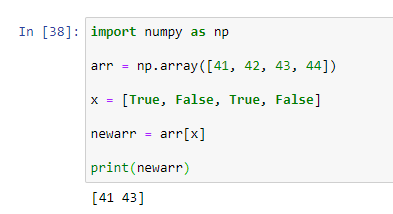
**Filter Arrays**

**Filtering Arrays:**

Getting some elements out of an existing array and creating a new array out of them is called filtering.

In NumPy, you filter an array using a boolean index list. A boolean index list is a list of booleans corresponding to indexes in the array. If the value at an index is Truethat element is contained in the filtered array, if the value at that index is False that element is excluded from the filtered array.

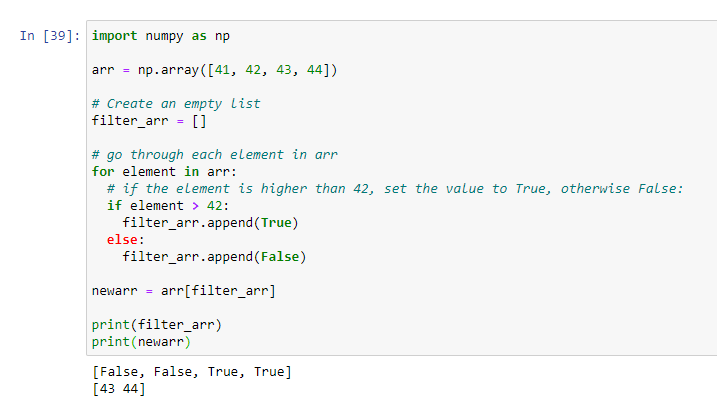
**Create an array from the elements on index 0 and 2:**



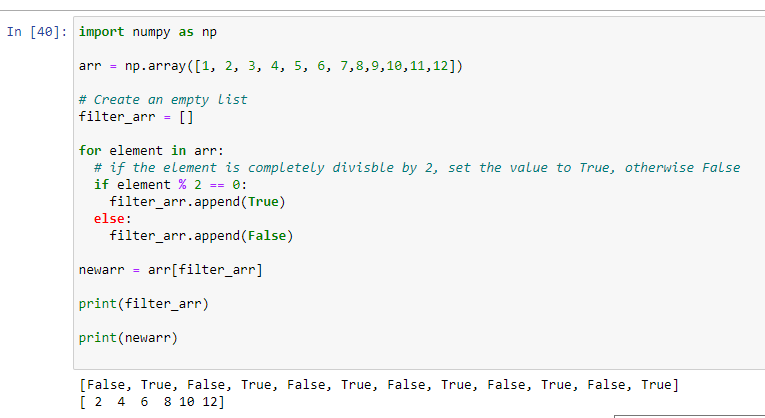
The example above will return [41, 43], Because the new filter contains only the values where the filter array had the value **True**, in this case, index 0 and 2.

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions. Here giving the condition to use the filtering arrays

Create a filter array that will return only values higher than 42.



Create a filter array that will return only even elements from the original array:



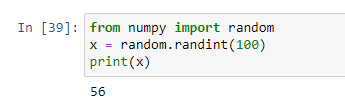
**Random**

Random number does not mean a different number every time. Random means something that cannot be predicted logically.

1. **Generate Random Number**

NumPy offers the random module to work with random numbers.

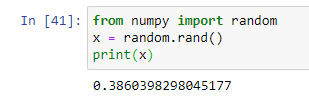
Generate a random integer from 0 to 100:



1. **Generate Random Float**

The random module's rand() method returns a random float between 0 and 1.

Generate a random float from 0 to 1:



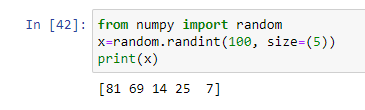
1. **Generate Random Array**

In NumPy we work with arrays, and you can use the two methods from the above examples to make random arrays.

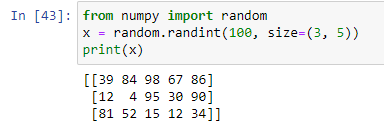
1. **Integers**

The randint() method takes a size parameter where you can specify the shape of an array.

Generate a 1-D array containing 5 random integers from 0 to 100:



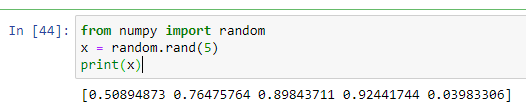
Generate a 2-D array with 3 rows, each row containing 5 random integers from 0 to 100:



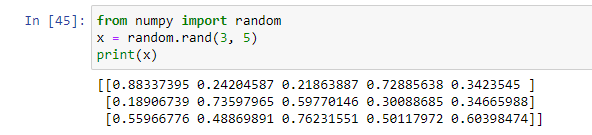
1. **Floats**

The rand() method also allows you to specify the shape of the array.

Generate a 1-D array containing 5 random floats:

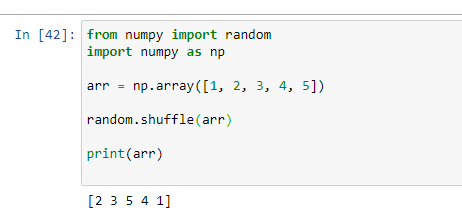


Generate a 2-D array with 3 rows, each row containing 5 random num



**Shuffling Arrays**

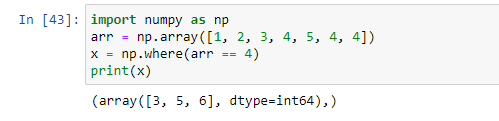
Shuffle means changing arrangement of elements in-place. i.e. in the array itself.



**Searching Arrays**

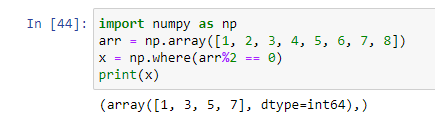
You can search an array for a certain value, and return the indexes that get a match. To search an array, use the **where()** method.

1. **Find the indexes where the value is 4:**

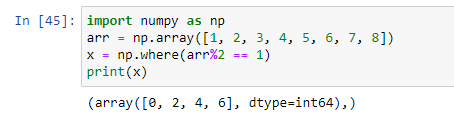


In the above example; the value 4 is present at index 3, 5, and 6.

1. **Find the indexes where the values are even:**



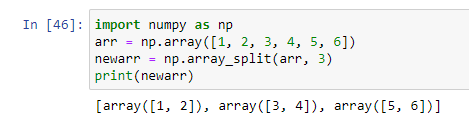
1. **Find the indexes where the values are odd:**



**Splitting NumPy Arrays**

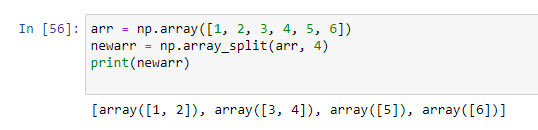
Splitting is reverse operation of Joining. Joining merges multiple arrays into one and Splitting breaks one array into multiple. We use **array\_split()** for splitting arrays, we pass it the array we want to split and the number of splits.

1. **Split the array in 3 parts:**



**Note:** The return value is an array containing three arrays. If the array has less elements than required, it will adjust from the end accordingly.

1. **Split the array in 4 parts:**



**Note:** We also have the method split() available but it will not adjust the elements when elements are less in source array for splitting like in example above, array\_split() worked properly but split() would fail.

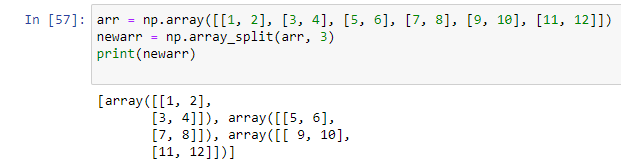
**Split Into Arrays**

The return value of the array\_split() method is an array containing each of the split as an array. If you split an array into 3 arrays, you can access them from the result just like any array elements

**Splitting 2-D Arrays**

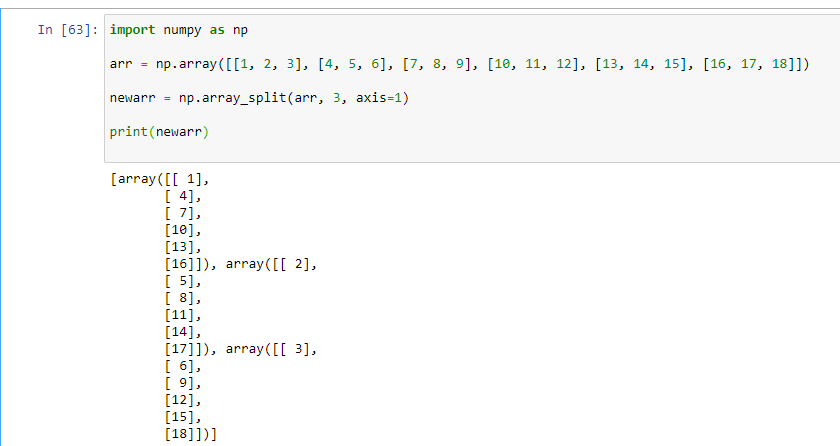
Use the same syntax when splitting 2-D arrays. Use the array\_split() method, pass in the array you want to split and the number of splits you want to do.

**Split the 2-D array into three 2-D arrays.**



The example above returns three 2-D arrays.

In addition, you can specify which axis you want to do the split around. The example below also returns three 2-D arrays, but they are split along the row (axis=1).



Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

The alternate solution for axis=1 is**; hsplit()**

