Numpy library

(Theory) What is NumPy?

NumPy is the fundamental package for scientific computing in Python. It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.

At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types, with many operations being performed in compiled code for performance. There are several important differences between NumPy arrays and the standard Python sequences:

NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically). Changing the size of an ndarray will create a new array and delete the original.

The elements in a NumPy array are all required to be of the same data type, and thus will be the same size in memory. The exception: one can have arrays of (Python, including NumPy) objects, thereby allowing for arrays of different sized elements.

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.

A growing plethora of scientific and mathematical Python-based packages are using NumPy arrays; though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays. In other words, in order to efficiently use much (perhaps even most) of today's scientific/mathematical Python-based software, just knowing how to use Python's built-in sequence types is insufficient - one also needs to know how to use NumPy arrays.

The points about sequence size and speed are particularly important in scientific computing. As a simple example, consider the case of multiplying each element in a 1-D sequence with the corresponding element in another sequence of the same length. If the data are stored in two Python lists, a and b, we could iterate over each element:

```
c = []
for i in range(len(a)):
    c.append(a[i]*b[i])
```

This produces the correct answer, but if a and b each contain millions of numbers, we will pay the price for the inefficiencies of looping in Python. We could accomplish the same task much more quickly in C by writing (for clarity we neglect variable declarations and initializations, memory allocation, etc.)

```
for (i = 0; i < rows; i++) {
    c[i] = a[i]*b[i];
 }
```

This saves all the overhead involved in interpreting the Python code and manipulating Python objects, but at the expense of the benefits gained from coding in Python. Furthermore, the coding work required increases with the dimensionality of our data. In the case of a 2-D array, for example, the C code (abridged as before) expands to

```
for (i = 0; i < rows; i++) {
    for (j = 0; j < columns; j++) {
     c[i][j] = a[i][j]*b[i][j];
     }
 }
```

NumPy gives us the best of both worlds: element-by-element operations are the “default mode” when an ndarray is involved, but the element-by-element operation is speedily executed by pre-compiled C code. In NumPy

```
c = a * b
```

1. Introduction to Numpy

There are 6 general mechanisms for creating arrays:

1. Conversion from other Python structures (i.e. lists and tuples)
2. Intrinsic NumPy array creation functions (e.g. arange, ones, zeros, etc.)
3. Replicating, joining, or mutating existing arrays
4. Reading arrays from disk, either from standard or custom formats
5. Creating arrays from raw bytes through the use of strings or buffers
6. Use of special library functions (e.g., random)

You can use these methods to create ndarrays or Structured arrays. This document will cover general methods for ndarray creation.

1.1 Converting Python sequences to NumPy Arrays

NumPy arrays can be defined using Python sequences such as lists and tuples. Lists and tuples are defined using [...] and (...), respectively. Lists and tuples can define ndarray creation:

- a list of numbers will create a 1D array

In [3]:
```
import numpy as np
```

In [5]:
```
a1D = np.array([1, 2, 3, 4])
a1D
```

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

In [6]:
```
a2D = np.array([[1, 2], [3, 4]])
a2D
```

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

In [7]:
```
a2D = np.array([[1, 2], [3, 4]])
a2D
```

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

When you use numpy.array to define a new array, you should consider the dtype of the elements in the array, which can be specified explicitly. This feature gives you more control over the underlying data structures and how the elements are handled in C/C++ functions. If you are not careful with dtype assignments, you can get unwanted overflow, as such

In [8]:
```
a = np.array([127, 128, 129], dtype=np.int8)
a
```

Out[8]:
```
array([ 127, -128, -127], dtype=int8)
```

1.2 Intrinsic NumPy array creation functions

NumPy has over 40 built-in functions for creating arrays as laid out in the Array creation routines (https://numpy.org/doc/stable/reference/routines.array-creation.html#routines-array-creation (https://numpy.org/doc/stable/reference/routines.array-creation.html#routines-array-creation)). These functions can be split into roughly three categories, based on the dimension of the array they create:

- 1D arrays
- 2D arrays
- ndarrays

The 1D array creation functions e.g. numpy.linspace and numpy.arange generally need at least two inputs, start and stop.

numpy.arange creates arrays with regularly incrementing values. Check the documentation for complete information and examples. A few examples are shown:

In [9]:
```
np.arange(10)
```

Out[9]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [10]:
```
np.arange(2, 10, dtype=float)
```

Out[10]:
```
array([2., 3., 4., 5., 6., 7., 8., 9.])
```

In [11]:
```
np.arange(2, 3, 0.1)
```

Out[11]:
```
array([2. , 2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7, 2.8, 2.9])
```

1.3 2D array creation functions

The 2D array creation functions e.g. numpy.eye, numpy.diag, and numpy.vander define properties of special matrices represented as 2D arrays.

np.eye(n, m) defines a 2D identity matrix. The elements where i=j (row index and column index are equal) are 1 and the rest are 0, as such:

In [12]:
```
np.eye(3)
```

Out[12]:
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [13]:
```
np.eye(3, 5)
```

Out[13]:
```
array([[1., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0.],
       [0., 0., 1., 0., 0.]])
```

numpy.diag can define either a square 2D array with given values along the diagonal or if given a 2D array returns a 1D array that is only the diagonal elements. The two array creation functions can be helpful while doing linear algebra, as such:

In [14]:
```
np.diag([1, 2, 3])
```

Out[14]:
```
array([[1, 0, 0],
       [0, 2, 0],
       [0, 0, 3]])
```

In [15]:
```
np.diag([1, 2, 3], 1)
```

Out[15]:
```
array([[0, 1, 0, 0],
       [0, 0, 2, 0],
       [0, 0, 0, 3],
       [0, 0, 0, 0]])
```

In [16]:
```
a = np.array([[1, 2], [3, 4]])
a
```

Out[16]:
```
array([[1, 2],
       [3, 4]])
```

In [17]:
```
np.diag(a)
```

Out[17]:
```
array([1, 4])
```

1.4 General ndarray creation functions

The ndarray creation functions e.g. numpy.ones, numpy.zeros, and random define arrays based upon the desired shape. The ndarray creation functions can create arrays with any dimension by specifying how many dimensions and length along that dimension in a tuple or list.

numpy.zeros will create an array filled with 0 values with the specified shape. The default dtype is float64:

In [18]:
```
np.zeros((2, 3))
```

Out[18]:
```
array([[0., 0., 0.],
       [0., 0., 0.]])
```

In [19]:
```
np.zeros((2, 3, 2))
```

Out[19]:
```
array([[[0., 0.],
        [0., 0.],
        [0., 0.]],

       [[0., 0.],
        [0., 0.],
        [0., 0.]]])
```

numpy.ones will create an array filled with 1 values. It is identical to zeros in all other respects as such:

In [20]:
```
np.ones((2, 3)) #2 rows 3 col.
```

Out[20]:
```
array([[1., 1., 1.],
       [1., 1., 1.]])
```

In [21]:
```
np.ones((2, 3, 2)) #2 rows 3 col. 2 channels
```

Out[21]:
```
array([[[1., 1.],
        [1., 1.],
        [1., 1.]],

       [[1., 1.],
        [1., 1.],
        [1., 1.]]])
```

The random method of the result of default_rng will create an array filled with random values between 0 and 1. It is included with the numpy.random library. Below, two arrays are created with shapes (2,3) and (2,3,2), respectively. The seed is set to 42 so you can reproduce these pseudorandom numbers:

In [23]:
```
from numpy.random import default_rng
```

In [24]:
```
default_rng(42).random((2,3))
```

Out[24]:
```
array([[0.77395605, 0.43887844, 0.85859792],
       [0.69736803, 0.09417735, 0.97562235]])
```

In [25]:
```
default_rng(42).random((2,3,2))
```

Out[25]:
```
array([[[0.77395605, 0.43887844],
        [0.85859792, 0.69736803],
        [0.09417735, 0.97562235]],

       [[0.7611397 , 0.78606431],
        [0.12811363, 0.45038594],
        [0.37079802, 0.92676499]]])
```

numpy.indices will create a set of arrays (stacked as a one-higher dimensioned array), one per dimension with each representing variation in that dimension:

In [26]:
```
np.indices((3,3))
```

Out[26]:
```
array([[[0, 0, 0],
        [1, 1, 1],
        [2, 2, 2]],

       [[0, 1, 2],
        [0, 1, 2],
        [0, 1, 2]]])
```

1.5 Replicating, joining, or mutating existing arrays

Once you have created arrays, you can replicate, join, or mutate those existing arrays to create new arrays. When you assign an array or its elements to a new variable, you have to explicitly numpy.copy the array, otherwise the variable is a view into the original array. Consider the following example:

In [27]:
```
a = np.array([1, 2, 3, 4, 5, 6])
a
```

Out[27]:
```
array([1, 2, 3, 4, 5, 6])
```

In [28]:
```
b = a[:2]
```

In [29]:
```
b += 1
```

In [30]:
```
print('a =', a, '; b =', b)
```

```
a = [2 3 3 4 5 6] ; b = [2 3]
```

In this example, you did not create a new array. You created a variable, b that viewed the first 2 elements of a. When you added 1 to b you would get the same result by adding 1 to a[:2]. If you want to create a new array, use the numpy.copy array creation routine as such:

In [34]:
```
a = np.array([1, 2, 3, 4])
b = a[:2].copy()
b += 1
print('a = ', a, 'b = ', b)
```

```
a =  [1 2 3 4] b =  [2 3]
```

2. Indexing on ndarrays

ndarrays can be indexed using the standard Python x[obj] syntax, where x is the array and obj the selection. There are different kinds of indexing available depending on obj: basic indexing, advanced indexing and field access.

Most of the following examples show the use of indexing when referencing data in an array. The examples work just as well when assigning to an array. See Assigning values to indexed arrays for specific examples and explanations on how assignments work.

Note that in Python, x[(exp1, exp2, ..., expN)] is equivalent to x[exp1, exp2, ..., expN]; the latter is just syntactic sugar for the former.

Single element indexing works exactly like that for other standard Python sequences. It is 0-based, and accepts negative indices for indexing from the end of the array.

In [35]:
```
x = np.arange(10)
x[3]
```

Out[35]:
```
3
```

In [36]:
```
x[-3]
```

Out[36]:
```
7
```

In [41]:
```
x.shape #dimensions of np.array
```

Out[41]:
```
(2, 5)
```

In [39]:
```
x.shape = (2, 5)  # now x is 2-dimensional
x
```

Out[39]:
```
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

In [38]:
```
x[1, 3]
```

Out[38]:
```
8
```

In [40]:
```
x[1, -1]
```

Out[40]:
```
9
```

Note that if one indexes a multidimensional array with fewer indices than dimensions, one gets a subdimensional array. For example:

In [42]:
```
x[0]
```

Out[42]:
```
array([0, 1, 2, 3, 4])
```

That is, each index specified selects the array corresponding to the rest of the dimensions selected. In the above example, choosing 0 means that the remaining dimension of length 5 is being left unspecified, and that what is returned is an array of that dimensionality and size. It must be noted that the returned array is a view, i.e., it is not a copy of the original, but points to the same values in memory as does the original array. In this case, the 1-D array at the first position (0) is returned. So using a single index on the returned array, results in a single element being returned. That is:

In [43]:
```
x[0][2]
```

Out[43]:
```
2
```

So note that **x[0, 2] == x[0][2]** though the second case is more inefficient as a new temporary array is created after the first index that is subsequently indexed by 2.

2.1 Slicing and striding

Basic slicing extends Python's basic concept of slicing to N dimensions. Basic slicing occurs when obj is a slice object (constructed by start:stop:step notation inside of brackets), an integer, or a tuple of slice objects and integers.

The simplest case of indexing with N integers returns an array scalar representing the corresponding item. As in Python, all indices are zero-based: for the i-th index n_i, the valid range is $0 <= n_i < d_i$ where d_i is the i-th element of the shape of the array. Negative indices are interpreted as counting from the end of the array (i.e., if $n_i < 0$, it means $n_i + d_i$).

All arrays generated by basic slicing are always views of the original array.

The standard rules of sequence slicing apply to basic slicing on a per-dimension basis (including using a step index). Some useful concepts to remember include:

- The basic slice syntax is i:j:k where i is the starting index, j is the stopping index, and k is the step (k!=0). This selects the m elements (in the corresponding dimension) with index values i, i + k, …, i + (m - 1) k where m=q+(r!=0) and q and r are the quotient and remainder obtained by dividing j - i by k: j - i = q k + r, so that i + (m - 1) k < j. For example:

In [44]:
```
x = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
x[1:7:2]
```

Out[44]:
```
array([1, 3, 5])
```

- Negative i and j are interpreted as n + i and n + j where n is the number of elements in the corresponding dimension. Negative k makes stepping go towards smaller indices. From the above example:

In [46]:
```
x[-2:10]
```

Out[46]:
```
array([8, 9])
```

In [47]:
```
x[-3:3:-1]
```

Out[47]:
```
array([7, 6, 5, 4])
```

- Assume n is the number of elements in the dimension being sliced. Then, if i is not given it defaults to 0 for k > 0 and n - 1 for k < 0 . If j is not given it defaults to n for k > 0 and -n-1 for k < 0 . If k is not given it defaults to 1. Note that :: is the same as : and means select all indices along this axis. From the above example:

In [48]:
```
x[5:]
```

Out[48]:
```
array([5, 6, 7, 8, 9])
```

- If the number of objects in the selection tuple is less than N, then : is assumed for any subsequent dimensions. For example:

In [50]:
```
x = np.array([[[1],[2],[3]], [[4],[5],[6]]])
x
```

Out[50]:
```
array([[[1],
        [2],
        [3]],

       [[4],
        [5],
        [6]]])
```

In [51]:
```
x.shape
```

Out[51]:
```
(2, 3, 1)
```

In [52]:
```
x[1:2]
```

Out[52]:
```
array([[[4],
        [5],
        [6]]])
```

- An integer, i, returns the same values as i:i+1 except the dimensionality of the returned object is reduced by 1. In particular, a selection tuple with the p-th element an integer (and all other entries :) returns the corresponding sub-array with dimension N - 1. If N = 1 then the returned object is an array scalar. These objects are explained in Scalars.
- If the selection tuple has all entries : except the p-th entry which is a slice object i:j:k, then the returned array has dimension N formed by concatenating the sub-arrays returned by integer indexing of elements i, i+k, …, i + (m - 1) k < j,
- Basic slicing with more than one non-: entry in the slicing tuple, acts like repeated application of slicing using a single non-: entry, where the non-: entries are successively taken (with all other non-: entries replaced by :). Thus, x[ind1, ..., ind2,:] acts like x[ind1][..., ind2, :] under basic slicing.
- You may use slicing to set values in the array, but (unlike lists) you can never grow the array. The size of the value to be set in x[obj] = value must be (broadcastable to) the same shape as x[obj].
- A slicing tuple can always be constructed as obj and used in the x[obj] notation. Slice objects can be used in the construction in place of the [start:stop:step] notation. For example, x[1:10:5, ::-1] can also be implemented as obj = (slice(1, 10, 5), slice(None, None, -1)); x[obj] . This can be useful for constructing generic code that works on arrays of arbitrary dimensions.

Ellipsis expands to the number of : objects needed for the selection tuple to index all dimensions. In most cases, this means that the length of the expanded selection tuple is x.ndim. There may only be a single ellipsis present. From the above example:

In [56]:
```
x[..., 0]
```

Out[56]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

This is equivalent to:

In [57]:
```
x[:, :, 0]
```

Out[57]:
```
array([[1, 2, 3],
       [4, 5, 6]])
```

Each newaxis object in the selection tuple serves to expand the dimensions of the resulting selection by one unit-length dimension. The added dimension is the position of the newaxis object in the selection tuple. newaxis is an alias for None, and None can be used in place of this with the same result. From the above example:

2.2 Boolean array indexing

This advanced indexing occurs when obj is an array object of Boolean type, such as may be returned from comparison operators. A single boolean index array is practically identical to x[obj.nonzero()] where, as described above, obj.nonzero() returns a tuple (of length obj.ndim) of integer index arrays showing the True elements of obj. However, it is faster when obj.shape == x.shape.

If obj.ndim == x.ndim, x[obj] returns a 1-dimensional array filled with the elements of x corresponding to the True values of obj. The search order will be row-major, C-style. An index error will be raised if the shape of obj does not match the corresponding dimensions of x, regardless of whether those values are True or False.

A common use case for this is filtering for desired element values. For example, one may wish to select all entries from an array which are not NaN:

In [58]:
```
x = np.array([[1., 2.], [np.nan, 3.], [np.nan, np.nan]])
x
```

Out[58]:
```
array([[ 1.,  2.],
       [nan,  3.],
       [nan, nan]])
```

In [59]:
```
x[~np.isnan(x)]
```

Out[59]:
```
array([1., 2., 3.])
```

Or wish to add a constant to all negative elements:

In [61]:
```
x = np.array([1., -1., -2., 3])
x
```

Out[61]:
```
array([ 1., -1., -2.,  3.])
```

In [63]:
```
x[x < 0] += 20
x
```

Out[63]:
```
array([ 1., 19., 18.,  3.])
```

In general if an index includes a Boolean array, the result will be identical to inserting obj.nonzero() into the same position and using the integer array indexing mechanism described above. x[ind_1, boolean_array, ind_2] is equivalent to x[(ind_1,) + boolean_array.nonzero() + (ind_2,)].

If there is only one Boolean array and no integer indexing array present, this is straightforward. Care must only be taken to make sure that the boolean index has exactly as many dimensions as it is supposed to work with.

In general, when the boolean array has fewer dimensions than the array being indexed, this is equivalent to x[b, ...], which means x is indexed by b followed by as many : as are needed to fill out the rank of x. Thus the shape of the result is one dimension containing the

number of True elements of the boolean array, followed by the remaining dimensions of the

In [65]:
```
x = np.arange(35).reshape(5, 7)
x
```

Out[65]:
```
array([[ 0,  1,  2,  3,  4,  5,  6],
       [ 7,  8,  9, 10, 11, 12, 13],
       [14, 15, 16, 17, 18, 19, 20],
       [21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

In [67]:
```
b = x > 20
b
```

Out[67]:
```
array([[False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False],
       [False, False, False, False, False, False, False],
       [ True,  True,  True,  True,  True,  True,  True],
       [ True,  True,  True,  True,  True,  True,  True]])
```

In [68]:
```
b[:, 5]
```

Out[68]:
```
array([False, False, False,  True,  True])
```

In [69]:
```
x[b[:, 5]]
```

Out[69]:
```
array([[21, 22, 23, 24, 25, 26, 27],
       [28, 29, 30, 31, 32, 33, 34]])
```

Summary of the main operations

In [70]:
```
#Creating a NumPy Array:
import numpy as np
array = np.random.rand(10)
array
```

Out[70]:
```
array([0.80169987, 0.95052702, 0.52299139, 0.91268935, 0.96484423,
       0.31623217, 0.49060006, 0.41328947, 0.02889025, 0.72133903])
```

In [77]:
```
#Sum of Two Arrays:
array1 = np.random.randint(10, size=5)
print("array1\n", array1)
array2 = np.random.randint(10, size=5)
print("array2\n", array2)
result = array1 + array2
print("result\n", result)
```

```
array1
 [7 9 6 6 7]
array2
 [1 9 7 6 9]
result
 [ 8 18 13 12 16]
```

In [78]:
```
#Element-wise Multiplication:
result = array1 * array2
result
```

Out[78]:
```
array([ 7, 81, 42, 36, 63])
```

In [81]:
```
#Maximum Value in an Array:
max_value = np.max(array1)
max_value
```

Out[81]:
```
9
```

In [83]:
```
#Minimum Value in an Array:
min_value = np.min(array2)
min_value
```

Out[83]:
```
1
```

In [84]:
```
#Mean (Average) of an Array:
mean = np.mean(result)
mean
```

Out[84]:
```
45.8
```

In [85]:
```
#Standard Deviation of an Array:
std_dev = np.std(array2)
std_dev
```

Out[85]:
```
2.939387691339814
```

In [86]:
```
#Creating a 3x3 NumPy Matrix with Random Integers:
matrix = np.random.randint(0, 10, (3, 3))
matrix
```

Out[86]:
```
array([[1, 9, 0],
       [0, 9, 0],
       [6, 8, 2]])
```

In [88]:
```
#Transposing a NumPy Matrix:
transposed_matrix = np.transpose(matrix)
transposed_matrix
```

Out[88]:
```
array([[1, 0, 6],
       [9, 9, 8],
       [0, 0, 2]])
```

In [90]:
```
#Matrix Multiplication:
matrix1 = np.random.randint(0, 10, (3, 3))
matrix2 = np.random.randint(0, 10, (3, 3))
print("Matrix1\n", matrix1)
result_matrix = np.dot(matrix1, matrix2)
print("Matrix2\n", matrix2)
result_matrix
```

```
Matrix1
 [[1 2 6]
 [3 0 3]
 [4 4 2]]
Matrix2
 [[0 4 5]
 [1 7 4]
 [5 0 6]]
```

Out[90]:
```
array([[32, 18, 49],
       [15, 12, 33],
       [14, 44, 48]])
```

3. NumPy Array Iterating

Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python. If we iterate on a 1-D array it will go through each element one by one.

Iterate on the elements of the following 1-D array:

In [2]:
```
arr = np.array([1, 2, 3])
for x in arr:
    print(x)
```

```
1
2
3
```

In a 2-D array it will go through all the rows. Iterate on the elements of the following 2-D array:

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

for x in arr:
    print(x)
```

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

```
To return the actual values, the scalars, we have to iterate the arrays
in each dimension.
Iterate on each scalar element of the 2-D array:
```

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

for x in arr:
    for y in x:
        print(y)
```

```
1
2
3
4
5
6
```

In a 3-D array it will go through all the 2-D arrays. Iterate on the elements of the following 3-D array:

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

for x in arr:
    print(x)
```

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

To return the actual values, the scalars, we have to iterate the arrays in each dimension. Iterate down to the scalars:

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

for x in arr:
    for y in x:
        for z in y:
            print(z)
```

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

3.1 Iterating Arrays Using nditer()

The function **nditer()** is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, lets go through it with examples.

In basic for loops, iterating through each scalar of an array we need to use n for loops which can be difficult to write for arrays with very high dimensionality.

Iterate through the following 3-D array:

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

for x in np.nditer(arr):
    print(x)
```

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

3.2 Iterating With Different Step Size

We can use filtering and followed by iteration.

Iterate through every scalar element of the 2D array skipping 1 element:

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

for x in np.nditer(arr[:, ::2]):
    print(x)
```

```
1
3
5
7
```

3.3 Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one.

Sometimes we require corresponding index of the element while iterating, the ndenumerate() method can be used for those usecases.

Enumerate on following 1D arrays elements:

In [10]:
```
arr = np.array([1, 2, 3])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

```
(0,) 1
(1,) 2
(2,) 3
```

Enumerate on following 2D array's elements:

In [11]:
```
arr = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr):
    print(idx, x)
```

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

Insights into numpy methods
https://www.w3schools.com/python/numpy/default.asp (https://www.w3schools.com/python/numpy/default.asp)

Exercises

https://www.w3schools.com/quiztest/quiztest.asp?qtest=NUMPY (https://www.w3schools.com/quiztest/quiztest.asp?qtest=NUMPY)
https://www.w3schools.com/python/numpy/exercise.asp (https://www.w3schools.com/python/numpy/exercise.asp)

Other numpy methods

https://numpy.org/doc/stable/reference/routines.array-creation.html#routines-array-creation (https://numpy.org/doc/stable/reference/routines.array-creation.html#routines-array-creation)

https://cs231n.github.io/python-numpy-tutorial/#numpy (https://cs231n.github.io/python-numpy-tutorial/#numpy)

Other Exercises
https://www.kaggle.com/code/themlphdstudent/learn-numpy-numpy-50-exercises-and-solution (https://www.kaggle.com/code/themlphdstudent/learn-numpy-numpy-50-exercises-and-solution)

https://www.w3schools.com/python/numpy/exercise.asp?filename=exercise_creating_arrays1 (https://www.w3schools.com/python/numpy/exercise.asp?filename=exercise_creating_arrays1)

https://pynative.com/python-numpy-exercise/ (https://pynative.com/python-numpy-exercise/)