In [2]:
```
import numpy as np
```

Methods

- **asarray()** convert input to an ndarray
- **eye(N), identity(N)** create N x N square identity matrix(1s on the diagonal and 0s elsewhere)
- **np.subtract(arr1,arr2)** `arr1 - arr2`
- **np.power(arr1,arr2)** `arr1 ** arr2`
- **np.sum(arr, axis= `optional`)** `zero-length arrays have sum 0`
- **np.mean(arr, axis=" `optional`)** `zero-length arrays have Nan mean`
- **arr.any()** `also work with non-boolean arrays where non-zero elements are equal to 1`
- **arr.all()** `also work with non-boolean arrays where non-zero elements are equal to 1`
- **arr.sort(axis)** `in-place sort, axis will be in nummber directly not axis keyword required`
- **np.sort()** `np.sort returns a sorted copy of an array instead of modifying the array in-place.`
- **np.in1d()** `Compute a boolean array indicating whether each element of x is contained in y`

```
values = np.array([6, 0, 0, 3, 2, 5, 6])
np.in1d(values, [2, 3, 6])
array([ True, False, False, True, True, False, True], dtype=bool)
```

- **np.unique(arr)** `Compute the sorted, unique elements in arr`
- **np.intersect1d(arr1,arr2)** `Compute the sorted, common elements in arr1 and arr2`
- **np.union1d(arr1,arr2)** `Compute the sorted union of elements`
- **setdiff1d(arr1,arr2)** `Set difference, elements in arr1 that are not in arr2`
- **setxor1d(arr1,arr2)** `Set symmetric differences; elements that are in either of the arrays, but not both`
- **np.save("fileName.npy", arr)** `Arrays are saved by default in an uncompressed raw binary format with file extension .npy:`

```
arr = np.arange(10)
np.save('abc.npy', arr)
```

- **np.savez(fileName, arr1,arr1)** `You save multiple arrays in an uncompressed archive using np.savez and passing the arrays as keyword arguments:`

```
np.savez('array_archive.npz', a=arr, b=arr)
```

- **np.load(fileName)** `When loading an .npz file, you get back a dict-like object that loads the individual arrays lazily:`

```
arch = np.load('array_archive.npz')
arch['b']
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

- **np.savez_compressed(fileName.npz, arr1,arr2)** If your data compresses well, you may wish to use numpy.savez_compressed instead:

```
np.savez_compressed('arrays_compressed.npz', a=arr, b=arr)
```

- **np.dot(arr1,arr2)** @ works as infix operator>>> **arr1 @ arr2** A matrix product between a two-dimensional array and a suitably sized onedimensional array results in a one-dimensional array:

```
np.dot(x, np.ones(3))
array([ 6., 15.])`
```

- **The diag(arr,K)** function is used to extract a diagonal or construct a diagonal array.
- Return the diagonal (or off-diagonal) elements of a square matrix as a 1D array,
- or convert a 1D array into a square matrix with zeros on the off-diagonal
- K is diagonal and is optional parameter . The default is 0 . Use k>0 for diagonals above the main diagonal , and k<0 for diagonals below the main diagonal
- **np.trace()** `Compute the sum of the diagonal elements
- **np.ravel(arr, F/C)** ravel does not produce a copy of
- **np.flatten(arr, F/C)** always returns a copy of the data
- **np.conconcate((arr1,arr2), axis=0)** |||| **np.vstack((arr1,arr2))** |||| **np.row_stack((arr1,arr2))** row wise stack
- **np.conconcate((arr1,arr2), axis=1)** |||| **np.hstack((arr1,arr2))** |||| **np.column_stack((arr1,arr2))** columns wise stack
- **np.hsplit((arr, tukre))** column wise split karega
- **np.vsplit((arr, tukre))** row wise split karega
- **arr.repeat(times)** repeat replicates each element in an array some number of times, producing a larger array:

```
arr
array([0, 1, 2])
arr.repeat(3)
array([0, 0, 0, 1, 1, 1, 2, 2, 2])
```

- By default, if you pass an integer, each element will be repeated that number of times. If you pass an array of integers, each element can be repeated a different number of times:

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

- Multidimensional arrays can have their elements repeated along a particular axis.

```
arr = np.random.randn(2, 2)
arr
array([[-2.0016, -0.3718],
      [ 1.669 , -0.4386]])
arr.repeat(2, axis=0)
array([[-2.0016, -0.3718],
      [-2.0016, -0.3718],
      [ 1.669 , -0.4386],
      [ 1.669 , -0.4386]])
```

- Note that if no axis is passed, the array will be flattened first, which is likely not what you want:

```
arr = np.random.randn(2, 2)
arr
array([[-1.2035238 ,  0.05843907],
     [-1.08394347,  0.04839468]])
arr.repeat(2)
array([-1.2035238 , -1.2035238 ,  0.05843907,  0.05843907, -1.08394347,
     -1.08394347,  0.04839468,  0.04839468])
```

- Similarly, you can pass an array of integers when repeating a multidimensional array to repeat a given slice a different number of times:

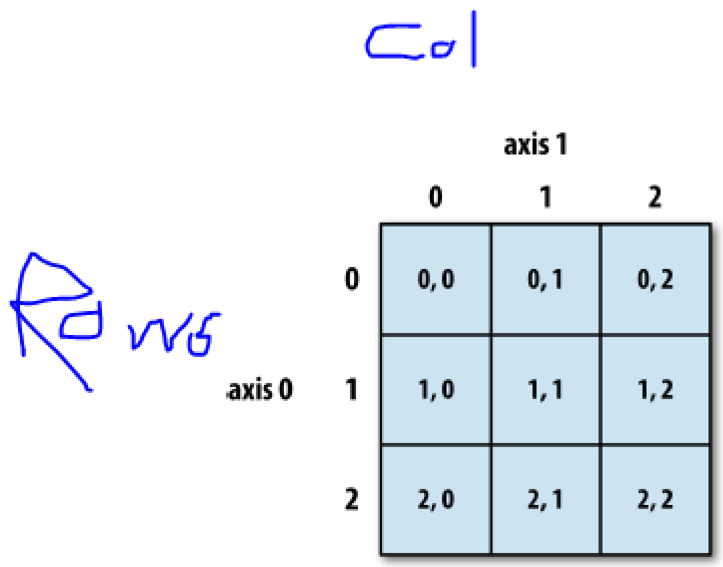
```
arr.repeat([2, 3], axis=0)
array([[-2.0016, -0.3718],
      [-2.0016, -0.3718],
      [ 1.669 , -0.4386],
      [ 1.669 , -0.4386],
      [ 1.669 , -0.4386]])
arr.repeat([2, 3], axis=1)
array([[-2.0016, -2.0016, -0.3718, -0.3718, -0.3718],
      [ 1.669 , 1.669 , -0.4386, -0.4386, -0.4386]])
```

- **np.tile(arr, axisNum)** tile is a shortcut for stacking copies of an array along an axis. The second argument is the number of tiles; with a scalar, the <b>tiling is made row by row</b>, rather than column by column.

```
In [62]: arr
Out[62]:
array([[-2.0016, -0.3718],
      [ 1.669 , -0.4386]])
In [63]: np.tile(arr, 2)
Out[63]:
```

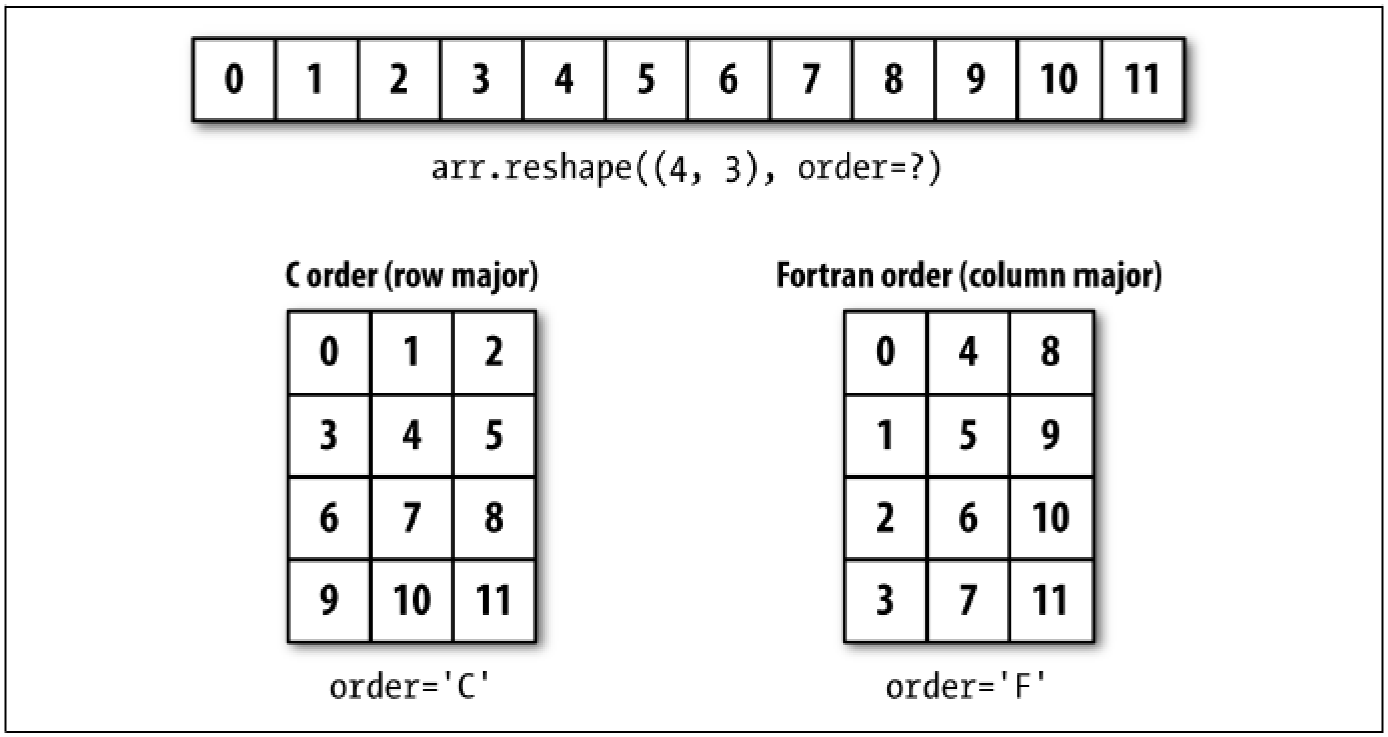
Theory

- **Calling** `astype` **always creates a** `new array (a copy of the data)` **, even if the new dtype is the same as the old dtype.**
- **vectorization**: **Any arithmetic operations** `between equal-size arrays` **applies the operation element-wise**
- **Comparisons between arrays of the same size yield boolean arrays**
- Operations between **differently sized** arrays is called **broadcasting**
- **array slices are `views` on the original array**.
- The **"bare" slice [:]** will assign to all values in an array: `arr_slice[:] = 45`
- **As NumPy has been designed to be able to work with very large arrays, you could imagine performance and memory problems if NumPy insisted on always copying data**
- In a **`2d array`, the `elements at each index` are no longer scalars but rather `1d arrays`**



- **The boolean array must be of the same length as the array axis it's indexing.**
- **Boolean selection** `will not fail` **if the boolean array is not the correct length, so I recommend care when using this feature`**
- **Selecting data from an array by boolean indexing** `always creates a copy of the data` **, even if the returned array is unchanged.**
- **Fancy indexing is a term to describe** `indexing using integer arrays`
- **the result of fancy indexing is always** `one-dimensional` **.**
- **Keep in mind that** `fancy indexing` **, unlike slicing,** `always copies` **the data into a new array.**

- **Boolean values are coerced to `1 (True)` and `0 (False)` in the preceding methods. Thus, `sum` is often used as a means of `counting True values in a boolean array`: `(arr > 0).sum() # Number of positive values`**
- **NumPy is able to save and load data to and from disk either in text or `binary format(we will learn this)`** ### Appendix section A2



- **One of the passed shape dimensions can be –1, in which case the value used for that dimension will be inferred from the data:**

```
arr = np.arange(15)
arr.reshape((5, -1))
array([[ 0, 1, 2],
        [ 3, 4, 5],
        [ 6, 7, 8],
        [ 9, 10, 11],
        [12, 13, 14]])
```

- **NumPy gives you control and flexibility `over the layout of your data in memory`. By `default`, NumPy arrays are created in `row major order`. This means that if you have a two-dimensional array of data, the items in each row of the array are stored in adjacent memory locations. The alternative to row major ordering is column major order, which means that values within each column of data are stored in adjacent memory locations.**

- **Functions like `reshape` and `ravel` accept an order argument indicating the order to use the data in the array. This is usually set to `'C'` or `'F'` in most cases (there are also `less commonly used options 'A' and 'K';`**

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

```
arr.ravel()
array([ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11])
```

```
arr.ravel('F')
array([ 0, 4, 8, 1, 5, 9, 2, 6, 10, 3, 7, 11])
```

- **The key difference between C and Fortran order is the way in which the dimensions are walked:**

C/row major order

Traverse higher dimensions `first` (e.g., axis 1 before advancing on axis 0).

Fortran/column major order

Traverse higher dimensions `last` (e.g., axis 0 before advancing on axis 1).

Three Dimensional data

- 3D data is a collection of 2D data-points(matrix). The shape of 3D data would be **(N,Row/Vector,Col/Scalar)**.
- **There would be N matrices of shape (M,P).**
- where **N** is the **number of matrices** in it and **Row/Vector** is the **number of vectors in each matrics**

```
In [3]: arr3d = np.array([[[1,1,1],[3,3,3]],[[2,2,2],[4,4,4]]])
        arr3d
```

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

               [[2, 2, 2],
                [4, 4, 4]]])
```

Applying sum function

- 1) Applying sum function across `axis-0` means you are `summing all matrices together`.
- 2) Applying sum function across `axis-1` means you are `summing all vectors inside each metrics`.
- 3) Applying sum function across `axis-2` means you are `summing all scalars inside each Vector`.

In [28]:
```
# 1)
np.sum(arr3d, axis=0)
```

Out[28]:
```
array([[3, 3, 3],
       [7, 7, 7]])
```

In [6]:
```
# 2)
np.sum(arr3d, axis=1)
```

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

In [7]:
```
# 3)
np.sum(arr3d, axis=2)
```

Out[7]:
```
array([[ 3,  9],
       [ 6, 12]])
```

In [19]:
```
x_test = np.arange(30).reshape(3, 2, 5)    #   (3,2,5)

print(x_test)
```

```
[[[ 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]]]
```

Transposing Arrays and swapping axis

- **Transposing is a special form of reshaping that similarly returns a `view` on the underlying data `without copying` anything. Arrays have the `transpose` method and also the special `T` attribute**
- **For higher dimensional arrays, transpose will accept a tuple of axis numbers to permute the axes**
- **ndarray has the method `swapaxes`, which takes a `pair of axis numbers and switches the indicated axes` to rearrange the data**

In [20]:
```
# np.transpose()
np.transpose(x_test,(0,1,2))   # (2,1,0), (2,0,1), (1,0,2), (1,2,0), (0,2,1)
```

Out[20]:
```
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]]])
```

In [21]:
```
print(x_test)
```

```
[[[ 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]]]
```

```
In [22]: np.transpose(x_test,(0,2,1))  # (3,2,5)  =>>>> (3,5,2)
```

```
Out[22]: array([[[ 0,  5],
                 [ 1,  6],
                 [ 2,  7],
                 [ 3,  8],
                 [ 4,  9]],

                [[10, 15],
                 [11, 16],
                 [12, 17],
                 [13, 18],
                 [14, 19]],

                [[20, 25],
                 [21, 26],
                 [22, 27],
                 [23, 28],
                 [24, 29]]])
```

```
In [24]: print(x_test)
         np.transpose(x_test,(2,0,1))  # (3,2,5)  =>>>> (5,3,2)
```

```
[[[ 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]]]
```

```
Out[24]: array([[[ 0,  5],
                 [10, 15],
                 [20, 25]],

                [[ 1,  6],
                 [11, 16],
                 [21, 26]],

                [[ 2,  7],
                 [12, 17],
                 [22, 27]],

                [[ 3,  8],
                 [13, 18],
                 [23, 28]],

                [[ 4,  9],
                 [14, 19],
                 [24, 29]]])
```

swapaxes()

- **takes a pair of axis numbers and switches the indicated axes to rearrange the data**
- **swapaxes returns a `view on the data without making a copy`.**

In [26]:
```
arr = np.arange(16).reshape((2, 2, 4))
arr
```

Out[26]:
```
array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

In [27]:
```
arr.swapaxes(1, 2)
```

Out[27]:
```
array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],

       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

In [31]:
```
arr1 = np.arange(1,11)
print(arr1)
print("================================")
arr2 = np.arange(11,21)
print(arr2)
```

```
[ 1  2  3  4  5  6  7  8  9 10]
================================
[11 12 13 14 15 16 17 18 19 20]
```

`np.greater()`

- Perform element-wise comparison, returning boolean array
- same as `>` operator in python

```
In [33]: np.greater(arr1,arr2)

Out[33]: array([False, False, False, False, False, False, False, False, False,
                False])
```

np.greater_equal()

- same as `>=` in python

```
In [36]: np.greater_equal(arr1,arr2)

Out[36]: array([False, False, False, False, False, False, False, False, False,
                False])
```

np.less()

- sames as `<` in python

```
In [38]: np.less(arr1,arr2)

Out[38]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True])
```

np.less_equal()

- same as `<=` in python

```
In [39]: np.less_equal(arr1,arr2)

Out[39]: array([ True,  True,  True,  True,  True,  True,  True,  True,  True,
                 True])
```

np.logical_and()

- Compute element-wise truth value of logical operation
- same as `&` in numpy

In [49]:
```
np.logical_and(True,False)
```
Out[49]:
```
False
```

In [50]:
```
np.logical_and([True, False], [False, False])
```
Out[50]:
```
array([False, False])
```

In [67]:
```
x = np.arange(5)
print(x)

x1 = x > 1
print(x1)
x2 = x < 4
print(x2)
```
```
[0 1 2 3 4]
[False False  True  True  True]
[ True  True  True  True False]
```

In [59]:
```
np.logical_and(x>1, x<4)
```
Out[59]:
```
array([False, False,  True,  True, False])
```

Array wise comparision

In [61]:
```
a = np.array([1,2,3,4])
b = np.array([2,4,6,8])
c = np.array([1,2,3,4])
```

```
In [62]: np.array_equal(a,b)
Out[62]: False
```

```
In [63]: np.array_equal(a,c)
Out[63]: True
```

Logical Operations

```
In [64]: a = np.array([1,1,0,0])
         b = np.array([1,0,1,0])
```

logical_or()

```
In [65]: np.logical_or(a,b)
Out[65]: array([ True,  True,  True, False])
```

logical_and()

```
In [66]: np.logical_and(a,b)
Out[66]: array([ True, False, False, False])
```

Stacking helpers: r *and c*

- There are two special objects in the NumPy namespace, r *and c*, that make stacking arrays more concise:

```
In [3]: arr = np.arange(6)
        arr

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

```
In [4]: arr1 = arr.reshape((3, 2))
        arr1

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

```
In [5]: arr2 = np.random.randn(3, 2)
        arr2

Out[5]: array([[ 0.08233373, -1.0915393 ],
               [-0.49037529,  0.85509949],
               [ 2.06904791, -1.19188202]])
```

```
In [6]: np.r_[arr1, arr2]

Out[6]: array([[ 0.        ,  1.        ],
               [ 2.        ,  3.        ],
               [ 4.        ,  5.        ],
               [ 0.08233373, -1.0915393 ],
               [-0.49037529,  0.85509949],
               [ 2.06904791, -1.19188202]])
```

```
In [9]: np.c_[np.r_[arr1, arr2], arr]

Out[9]: array([[ 0.        ,  1.        ,  0.        ],
               [ 2.        ,  3.        ,  1.        ],
               [ 4.        ,  5.        ,  2.        ],
               [ 0.08233373, -1.0915393 ,  3.        ],
               [-0.49037529,  0.85509949,  4.        ],
               [ 2.06904791, -1.19188202,  5.        ]])
```

In [10]:
```
# These additionally can translate slices to arrays:
np.c_[1:6, -10:-5]
```

Out[10]:
```
array([[  1, -10],
       [  2,  -9],
       [  3,  -8],
       [  4,  -7],
       [  5,  -6]])
```

Fancy Indexing Equivalents: take and put

- As you may recall from Chapter 4, one way to get and set subsets of arrays is by fancy indexing using integer arrays:

In [13]:
```
arr = np.arange(10) * 100
arr
```

Out[13]:
```
array([  0, 100, 200, 300, 400, 500, 600, 700, 800, 900])
```

In [14]:
```
inds = [7, 1, 2, 6]
```

In [15]:
```
arr[inds]
```

Out[15]:
```
array([700, 100, 200, 600])
```

There are alternative ndarray methods that are useful in the special case of `only making a selection on a single axis`:

In [16]:
```
arr.take(inds)
```

Out[16]:
```
array([700, 100, 200, 600])
```

In [17]:
```
arr.put(inds, 42)
```

In [18]:
```
arr
```

Out[18]:
```
array([  0,  42,  42, 300, 400, 500,  42,  42, 800, 900])
```

```
In [19]: arr.put(inds, [40, 41, 42, 43])
```

```
In [20]: arr
```

```
Out[20]: array([  0,  41,  42, 300, 400, 500,  43,  40, 800, 900])
```

To use `take` along other axes , you can pass the `axis` keyword :

```
In [21]: inds = [2, 0, 2, 1]
```

```
In [28]: arr = np.random.randn(2, 4)
         arr
```

```
Out[28]: array([[-0.41084115,  0.63376438,  1.07303783, -1.91632259],
                [ 1.5397761 ,  0.72241034,  0.57322275,  1.05599246]])
```

```
In [29]: arr.take(inds, axis=1)
```

```
Out[29]: array([[ 1.07303783, -0.41084115,  1.07303783,  0.63376438],
                [ 0.57322275,  1.5397761 ,  0.57322275,  0.72241034]])
```

put does not accept an axis argument but rather indexes into the flattened (onedimensional, C order) version of the array.