Data mining Technique Using Python/R

Avijit Bose
Assistant Professor
Department of Computer Science & Engineering,
MCKV institute of Engineering,Liluah,Howrah-711204

Working with Numpy

- Preexisting is the installation of numpy packages along with matplotlib.
- Numpy deals with homogeneous array.
- It is a table of elements indexed by non negative numbers.
- In numpy dimensions are called axis.
- For example if we write the following :-
- [[1,0,0],[0,1,1]] this has got 2 axis 1st axis has got 2 rows and 2nd axis has got 3 columns which in python words are said as number of elements it consists of.

Often asked in the interview

- Difference between numpy ndarray and array?

-> Numpy's array class is called ndarray and we can write it as numpy.array but it is not the same as python library class array.array which can consist of only 1 dimensional arrays and offer less functionality.

Some of the properties of ndarray are shown in the next slide:-

ndarray properties

- ndarray.ndim:- returns the number of axes or dimensions of the array.
- ndarray.shape:- It represents the dimensions of the array. This is a tuple of integers indicating the size of the array in each dimension. For a matrix with n rows and m columns shape will be (n,m). The length of the shape of the tuple is therefore the number of axes ndim.
- ndarray.size:- The total number of elements of the array . This is equal to the product of the elements of shape.
- ndarray.dtype:- an object describing the type of the elements in the array. One can create or specify dtype's using standard Python types. Additionally NumPy provides types of its own. numpy.int32, numpy.int16, and numpy.float64 are some examples.
- ndarray.itemsize:- the size in bytes of each element of the array. For example, an array of elements of type float64 has itemsize 8 (=64/8), while one of type complex32 has itemsize 4 (=32/8). It is equivalent to ndarray.dtype.itemsize.

ndarray properties

- ndarray.data :- the buffer containing the actual elements of the array. Normally, we won’t need to use this attribute because we will access the elements in an array using indexing facilities.
- Now lets see an example:-

Example

```
• >>import numpy as np
• >> a = np.arange(15).reshape(3, 5)
•    a
• array([[ 0,  1,  2,  3,  4],
•        [ 5,  6,  7,  8,  9],
•        [10, 11, 12, 13, 14]])
• >> a.shape
• (3, 5)
•  >> a.ndim
• 2
• >>a.dtype.name
• 'int64'
```

Example

- >> a.itemsize
- 8
- >> a.size
- 15
- >> type(a)
- <class 'numpy.ndarray'>
- >> b = np.array([6, 7, 8])
- >>   b
- array([6, 7, 8])
- >> type(b)
- <class 'numpy.ndarray'>

Array creation

- How to create an array
- >>> import numpy as np
- >>> a = np.array([2,3,4])
- >>> a
- array([2, 3, 4])
- >>> a.dtype
- dtype('int64')
- >>> b = np.array([1.2, 3.5, 5.1])
- >>> b.dtype
- dtype('float64')
- For example we can create an array from a regular python list or tuple using python list or tuple using the array function.

Where the error is faced?

- `>>> a = np.array(1,2,3,4)    # WRONG`
- `Traceback (most recent call last):`
- `  ...`
- `TypeError: array() takes from 1 to 2 positional arguments but 4 were given`
- `>>> a = np.array([1,2,3,4])  # RIGHT`

ndarray

- Array transforms sequence of sequence into 2 dimensional sequences of 3 dimensional arrays
- >>> b = np.array([(1.5,2,3), (4,5,6)])
- >>> b
- array([[1.5, 2. , 3.],
-         [4. , 5. , 6.]])

ndarray

- The type of the ndarray can also be specified at creation time
- >>> c = np.array([[1,2], [3,4]], dtype=complex)
- >>> c
- array([[1.+0.j, 2.+0.j],
- [3.+0.j, 4.+0.j]])

ndarray

- The function zeros creates an array full of zeros, the function ones creates an array full of ones, and the function empty creates an array whose initial content is random and depends on the state of the memory. By default, the dtype of the created array is float64.

ndarray example

```
>>> np.zeros((3, 4))
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
>>> np.ones( (2,3,4), dtype=np.int16 )                # dtype can also be specified
array([[[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]],

       [[1, 1, 1, 1],
        [1, 1, 1, 1],
        [1, 1, 1, 1]]], dtype=int16)
>>> np.empty( (2,3) )                                 # uninitialized
array([[  3.73603959e-262,   6.02658058e-154,   6.55490914e-260],  # may vary
       [  5.30498948e-313,   3.14673309e-307,   1.00000000e+000]])
```

ndarray

- To create sequences of numbers, NumPy provides the arange function which is analogous to the Python built-in range, but returns an array.
- >>> np.arange(10, 30, 5)
- array([10, 15, 20, 25])
- >>> np.arange(0, 2, 0.3)
- # it accepts float arguments
- array([0. , 0.3, 0.6, 0.9, 1.2, 1.5, 1.8])

ndarray

- When **arange** is used with floating point arguments, it is generally not possible to predict the number of elements obtained, due to the finite floating point precision. For this reason, it is usually better to use the function **linspace** that receives as an argument the number of elements that we want, instead of the step:

Example

- >>> from numpy import pi
- >>> np.linspace(0, 2, 9)
-   # 9 numbers from 0 to 2
- array([0. , 0.25, 0.5 , 0.75, 1. , 1.25, 1.5 , 1.75, 2.])
- >>> x = np.linspace(0, 2*pi, 100)
- # useful to evaluate function at lots of points
- >>> f = np.sin(x)

Printing Arrays

- When we print an array, NumPy displays it in a similar way to nested lists, but with the following layout:
- the last axis is printed from left to right,
- the second-to-last is printed from top to bottom,
- the rest are also printed from top to bottom, with each slice separated from the next by an empty line.

Printing Arrays

- One-dimensional arrays are then printed as rows, bidimensionals as matrices and tridimensionals as lists of matrices.

Printing Arrays

```
>>> a = np.arange(6)                    # 1d array
>>> print(a)
[0 1 2 3 4 5]
>>>
>>> b = np.arange(12).reshape(4,3)      # 2d array
>>> print(b)
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
>>>
>>> c = np.arange(24).reshape(2,3,4)    # 3d array
>>> print(c)
[[[ 0  1  2  3]
  [ 4  5  6  7]
  [ 8  9 10 11]]

 [[12 13 14 15]
  [16 17 18 19]
  [20 21 22 23]]]
```

Reshape() method

```
•  >>> print(np.arange(10000))
•  [   0    1    2 ... 9997 9998 9999]
•  >>>
•  >>> print(np.arange(10000).reshape(100,100))
•  [[   0    1    2 ...   97   98   99]
•   [ 100  101  102 ...  197  198  199]
•   [ 200  201  202 ...  297  298  299]
•   ...
•   [9700 9701 9702 ... 9797 9798 9799]
•   [9800 9801 9802 ... 9897 9898 9899]
•   [9900 9901 9902 ... 9997 9998 9999]]
```

Basic Operations

- Arithmetic operators on arrays apply elementwise. A new array is created and filled with the result.

Operations on ndarray

```
>>> a = np.array( [20,30,40,50] )
>>> b = np.arange( 4 )
>>> b
array([0, 1, 2, 3])
>>> c = a-b
>>> c
array([20, 29, 38, 47])
>>> b**2
array([0, 1, 4, 9])
>>> 10*np.sin(a)
array([ 9.12945251, -9.88031624,  7.4511316 , -2.62374854])
>>> a<35
array([ True,  True, False, False])
```

Operations on ndarray

- Unlike in many matrix languages, the product operator * operates elementwise in NumPy arrays. The matrix product can be performed using the @ operator (in python >=3.5) or the dot function or method:

ndarray example

```
>>> A = np.array( [[1,1],
...               [0,1]] )
>>> B = np.array( [[2,0],
...               [3,4]] )
>>> A * B                      # elementwise product
array([[2, 0],
       [0, 4]])
>>> A @ B                      # matrix product
array([[5, 4],
       [3, 4]])
>>> A.dot(B)                   # another matrix product
array([[5, 4],
       [3, 4]])
```

ndarray operations

- Some operations, such as += and *=, act in place to modify an existing array rather than create a new one.

Example of ndarray example

- >>> rg = np.random.default _ rng(1)
- # create instance of default random number generator
- >>> a = np.ones((2,3), dtype=int)
- >>> b = rg.random((2,3))
- >>> a *= 3
- >>> a
- array([[3, 3, 3],
-        [3, 3, 3]])
- >>> b += a
- >>> b
- array([[3.51182162, 3.9504637 , 3.14415961],
-        [3.94864945, 3.31183145, 3.42332645]])
- >>> a += b
- # b is not automatically converted to integer type
- Traceback (most recent call last):
-    ...
- numpy.core._exceptions._UFuncOutputCastingError: Cannot cast ufunc 'add' output from dtype('float64') to dtype('int64') with casting rule 'same_kind

ndarray

```
>>> a = np.ones(3, dtype=np.int32)
>>> b = np.linspace(0,pi,3)
>>> b.dtype.name
'float64'
>>> c = a+b
>>> c
array([1.        , 2.57079633, 4.14159265])
>>> c.dtype.name
'float64'
>>> d = np.exp(c*1j)
>>> d
array([ 0.54030231+0.84147098j, -0.84147098+0.54030231j,
       -0.54030231-0.84147098j])
>>> d.dtype.name
'complex128'
```

ndarray

- Many unary operations, such as computing the sum of all the elements in the array, are implemented as methods of the ndarray class.

ndarray

- >>> a = rg.random((2,3))
- >>> a
- array([[0.827770259, 0.40919914, 0.54959369],
-        [0.02755911, 0.75351311, 0.53814331]])
- >>> a.sum()
- 3.1057109529998157
- >>> a.min()
- 0.027559113243068367
- >>> a.max()
- 0.8277025938204418

Axis parameter

- By default, these operations apply to the array as though it were a list of numbers, regardless of its shape. However, by specifying the axis parameter you can apply an operation along the specified axis of an array:

Axis parameter

```
>>> b = np.arange(12).reshape(3,4)
>>> b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
>>>
>>> b.sum(axis=0)                         # sum of each column
array([12, 15, 18, 21])
>>>
>>> b.min(axis=1)                         # min of each row
array([0, 4, 8])
>>>
>>> b.cumsum(axis=1)                         # cumulative sum along each row
array([[ 0,  1,  3,  6],
       [ 4,  9, 15, 22],
       [ 8, 17, 27, 38]])
```

Universal Functions

- NumPy provides familiar mathematical functions such as sin, cos, and exp. In NumPy, these are called “universal functions”(ufunc). Within NumPy, these functions operate elementwise on an array, producing an array as output.

Universal functions

- >>> B = np.arange(3)
- >>> B
- array([0, 1, 2])
- >>> np.exp(B)
- array([1.   , 2.71828183, 7.3890561])
- >>> np.sqrt(B)
- array([0.   , 1.   , 1.41421356])
- >>> C = np.array([2., -1., 4.])
- >>> np.add(B, C)
- array([2., 0., 6.])

Indexing, Slicing,Iterating

- One dimensional arrays can be indexed, sliced and iterated over much like lists and other python sequences.

Example

- >>> a = np.arange(10)**3
- >>> a
- array([0,   1,   8, 27, 64, 125, 216, 343, 512, 729])
- >>> a[2]
- 8
- >>> a[2:5]
- array([8, 27, 64])

Example

- # equivalent to a[0:6:2] = 1000;
- # from start to position 6, exclusive, set every 2nd element to 1000
- >>> a[:6:2] = 1000
- >>> a
- array([1000,   1, 1000,   27, 1000, 125, 216, 343, 512, 729])
- >>> a[: :-1]   # reversed a
- array([729, 512, 343, 216, 125, 1000,   27, 1000,   1, 1000])
- >>> for i in a:
- ...   print(i**(1/3.))
- ...
- 9.999999999999998
- 1.0
- 9.999999999999998
- 3.0
- 9.999999999999998
- 4.999999999999999
- 5.999999999999999
- 6.999999999999999
- 7.999999999999999
- 8.999999999999998

Multidimensional array concept

```
>>> def f(x,y):
...     return 10*x+y
...
>>> b = np.fromfunction(f,(5,4),dtype=int)
>>> b
array([[ 0,  1,  2,  3],
       [10, 11, 12, 13],
       [20, 21, 22, 23],
       [30, 31, 32, 33],
       [40, 41, 42, 43]])
>>> b[2,3]
23
```

Multidimensional array concept

- >>> b[0:5, 1] # each row in the second column of b
- array([1, 11, 21, 31, 41])
- >>> b[: ,1]   # equivalent to the previous example
- array([1, 11, 21, 31, 41])
- >>> b[1:3, :]   # each column in the second and third row of b
- array([[10, 11, 12, 13],
-        [20, 21, 22, 23]])

Multidimensional array and slicing

- When fewer indices are provided than the number of axes, the missing indices are considered complete slices:

- >>> b[-1]   # the last row. Equivalent to b[-1,:]
- array([40, 41, 42, 43])

Multidimensional array & slicing

- The expression within brackets in b[i] is treated as an i followed by as many instances of : as needed to represent the remaining axes. NumPy also allows you to write this using dots as b[i,...].

- The dots (...) represent as many colons as needed to produce a complete indexing tuple. For example, if x is an array with 5 axes, then

- x[1,2,...] is equivalent to x[1,2,:,:,:],

- x[...,3] to x[:,:,:,:,3] and

- x[4,...,5,:] to x[4,:,:,5,:].

Multidimensional array and slicing

```
>>> c = np.array( [[[  0,  1,  2],     # a 3D array (two stacked 2D arrays)
...                [ 10, 12, 13]],
...               [[100,101,102],
...                [110,112,113]]])
>>> c.shape
(2, 2, 3)
>>> c[1,...]                           # same as c[1,:,:] or c[1]
array([[100, 101, 102],
       [110, 112, 113]])
>>> c[...,2]                           # same as c[:,:,2]
array([[  2,  13],
       [102, 113]])
```

Iteration

- Iterating over multidimensional arrays is done with respect to the first axis:
- `>>> for row in b:`
- `        print(row)`
- However, if one wants to perform an operation on each element in the array, one can use the flat attribute which is an iterator over all the elements of the array:
- `>>> for element in b.flat:`

  `        print(element)`

Shape and Reshape

- An array has a shape given by the number of elements along each axis.
- `>>> a = np.floor(10*rg.random((3,4)))`
- `>>> a`
- `array([[3., 7., 3., 4.],`
- `       [1., 4., 2., 2.],`
- `       [7., 2., 4., 9.]])`
- `>>> a.shape`
- `(3, 4)`

Shape and Reshape

- The shape of an array can be changed with various commands. Note that the following three commands all return a modified array, but do not change the original array:

Shape & Reshape

- >>> a.ravel() # returns the array, flattened
- array([3., 7., 3., 4., 1., 4., 2., 2., 7., 2., 4., 9.])
- >>> a.reshape(6,2) # returns the array with a modified shape
- array([[3., 7.],
-        [3., 4.],
-        [1., 4.],
-        [2., 2.],
-        [7., 2.],
-        [4., 9.]])
- >>> a.T # returns the array, transposed
- array([[3., 1., 7.],
-        [7., 4., 2.],
-        [3., 2., 4.],
-        [4., 2., 9.]])
- >>> a.T.shape
- (4, 3)
- >>> a.shape
- (3, 4)

Shape and Reshape

- The reshape function returns its argument with a modified shape, whereas the ndarray.resize method modifies the array itself:
- >>> a
- array([[3., 7., 3., 4.],
-         [1., 4., 2., 2.],
-         [7., 2., 4., 9.]])
- >>> a.resize((2,6))
- >>> a
- array([[3., 7., 3., 4., 1., 4.],
-         [2., 2., 7., 2., 4., 9.]])

Shape and reshape

- If a dimension is given as -1 in a reshaping operation, the other dimensions are automatically calculated.
- >>> a.reshape(3,-1)
- array([[3., 7., 3., 4.],
-        [1., 4., 2., 2.],
-        [7., 2., 4., 9.]])

Copies and Views

- No copy at all
- Simple assignments make no copy of the data at all.
- `>>> a = np.array([[ 0,  1,  2,  3],`
- `...                [ 4,  5,  6,  7],`
- `...                [ 8,  9, 10, 11]])`
- `>>> b = a            # no new object is created`
- `>>> b is a     # a and b are two names for the same ndarray object`
- True

Copies

- Python passes mutable objects as references, so function calls make no copy.
- >>> def f(x):
- ...   print(id(x))
- ...
- >>> id(a)   # id is a unique identifier of an object
- 148293216 # may vary
- >>> f(a)
- 148293216 # may vary

View or shallow copy

- Different array objects can share the same data. The view method creates a new array object that looks at the same data.
- >>> c = a.view()
- >>> c is a
- False
- >>> c.base is a # c is a view of the data owned by a
- True

View or shallow copy

- >>> c.flags.owndata
- False
- >>>
- >>> c = c.reshape((2, 6))   # a's shape doesn't change
- >>> a.shape
- (3, 4)
- >>> c[0, 4] = 1234   # a's data changes
- >>> a
- array([[0,   1,   2,   3],
-        [1234,   5,   6,   7],
-        [8,   9,   10,   11]])

Deep Copy

- The copy method makes a complete copy of the array and its data.
- >>> d = a.copy()   # a new array object with new data is created
- >>> d is a
- False
- >>> d.base is a   # d doesn't share anything with a
- False
- >>> d[0,0] = 9999
- >>> a
- array([[0,   10,   10,   3],
-        [1234,   10,   10,   7],
-        [8,   10,   10,   11]])

Deep Copy

- Sometimes copy should be called after slicing if the original array is not required anymore. For example, suppose a is a huge intermediate result and the final result b only contains a small fraction of a, a deep copy should be made when constructing b with slicing:

Deep Copy

- >>> a = np.arange(int(1e8))
- >>> b = a[:100].copy()
- >>> del a # the memory of ``a`` can be released.
- If b = a[:100] is used instead, a is referenced by b and will persist in memory even if del a is executed.

Advanced indexing tricks

- Numpy offers more indexing capabilities than regular python sequence.
- Arrays can be indexed by array of integers and array of Booleans
- Suppose we write the following:-
- `>>> a= np.arange(20)**3`
- `>>> i=np.array([1,1,3,8,5])`
- `>>> a[i]= array([ 1,  1,  9, 64, 25])`
- `>>> j = np.array([[3, 4], [9, 7]])`
- `>>> a[j]`
- `array([[ 9, 16],`
- `       [81, 49]])`

Advanced Indexing tricks

- >>> a = np.arange(12).reshape(3,4)
- >>> a
- array([[0, 1, 2, 3],
-        [4, 5, 6, 7],
-        [8, 9, 10, 11]])
- i = np.array([[0, 1],
-                     [1, 2]])
- >>> j = np.array([[2, 1],
-                          [3, 3]])
- Now what does the above statement means this is very important. This means
- 0,2 1,1
- 1,3,   2,3   so now put the values you should get
- 2 5
- 7 11
-

Advanced Slicing Techniques

```
• >>> a[i, j]                    # i and j must have equal shape
• array([[ 2,  5],
•        [ 7, 11]])
• >>> a[i, 2]
• array([[ 2,  6],
•        [ 6, 10]])
• >>> a[:, j]                     # i.e., a[ : , j]
• array([[[ 2,  1],
•         [ 3,  3]],

•        [[ 6,  5],
•         [ 7,  7]],

•        [[10,  9],
•         [11, 11]]])
```

Concept of tuple

- In Python, arr[i, j] is exactly the same as arr[(i, j)]—so we can put i and j in a tuple and then do the indexing with that.
- >>> l = (i, j)
- # equivalent to a[i, j]
- >>> a[l]
- array([[2, 5],
-         [7, 11]])

Advance slicing technique

- However, we can not do this by putting i and j into an array, because this array will be interpreted as indexing the first dimension of a.
- `>>> s = np.array([i, j])`

- `# not what we want`
- `>>> a[s]`
- `Traceback (most recent call last):`
- `  File "<stdin>", line 1, in <module>`
- `IndexError: index 3 is out of bounds for axis 0 with size 3`

- `# same as a[i, j]`
- `>>> a[tuple(s)]`
- `array([[ 2,  5],`
- `       [ 7, 11]])`

Advance slicing technique

- >>> time = np.linspace(20, 145, 5)   # time scale
- >>> data = np.sin(np.arange(20)).reshape(5,4)   # 4 time-dependent series
- >>> time
- array([20. , 51.25, 82.5 , 113.75, 145.])
- >>> data
- array([[0.   , 0.84147098, 0.90929743, 0.14112001],
-        [-0.7568025 , -0.95892427, -0.2794155 , 0.6569866],
-        [0.98935825, 0.41211849, -0.54402111, -0.99999021],
-        [-0.53657292, 0.42016704, 0.99060736, 0.65028784],
-        [-0.28790332, -0.96139749, -0.75098725,0.14987721]])

Advance slicing technique

- >>> ind = data.argmax(axis=0)
- >>> ind
- array([2, 0, 3, 1])

- # times corresponding to the maxima
- >>> time_max = time[ind]
- >>>
- >>> data_max = data[ind, range(data.shape[1])]
- >>> time_max
- array([82.5 , 20. , 113.75, 51.25])
- >>> data_max
- array([0.98935825, 0.84147098, 0.99060736, 0.6569866])
- >>> np.all(data_max == data.max(axis=0))
- True

Advance slicing technique

- You can also use indexing with arrays as a target to assign to:
- `>>> a = np.arange(5)`
- `>>> a`
- `array([0, 1, 2, 3, 4])`
- `>>> a[[1,3,4]] = 0`
- `>>> a`
- `array([0, 0, 2, 0, 0])`

Advance slicing technique

- However, when the list of indices contains repetitions, the assignment is done several times, leaving behind the last value:
- >>> a = np.arange(5)
- >>> a[[0,0,2]]=[1,2,3]
- >>> a
- array([2, 1, 3, 3, 4])

Advance Slicing Technique

- This is reasonable enough, but watch out if you want to use Python’s += construct, as it may not do what you expect:
- >>> a = np.arange(5)
- >>> a[[0,0,2]]+=1
- >>> a
- array([1, 1, 3, 3, 4])
- Even though 0 occurs twice in the list of indices, the 0th element is only incremented once. This is because Python requires “a+=1” to be equivalent to “a = a + 1”.

Advanced slicing Techniques

- Indexing with boolean arrays
- When we index arrays with arrays of (integer) indices we are providing the list of indices to pick. With boolean indices the approach is different; we explicitly choose which items in the array we want and which ones we don’t.
- The most natural way one can think of for boolean indexing is to use boolean arrays that have the same shape as the original array:

Advanced Slicing techniques

- `>>> a = np.arange(12).reshape(3,4)`
- `>>> b = a > 4`
- `>>> b                    # b is a boolean with a's shape`
- `array([[False, False, False, False],`
- `       [False,  True,  True,  True],`
- `       [ True,  True,  True,  True]])`
- `>>> a[b]       # 1d array with the selected elements`
- `array([ 5,  6,  7,  8,  9, 10, 11])`

Advanced Slicing techniques

- This property can be very useful in assignments:
- >>> a[b] = 0# All elements of 'a' higher than 4 become 0
- >>> a
- array([[0, 1, 2, 3],
-        [4, 0, 0, 0],
-        [0, 0, 0, 0]])

Advanced Slicing Techniques

- The second way of indexing with booleans is more similar to integer indexing; for each dimension of the array we give a 1D boolean array selecting the slices we want:
  `>>> a = np.arange(12).reshape(3,4)`
- `>>> b1 = np.array([False,True,True])             # first dim selection`
- `>>> b2 = np.array([True,False,True,False])       # second dim selection`
- `>>>`
- `>>> a[b1,:]                                   # selecting rows`
- `array([[ 4,  5,  6,  7],`
- `       [ 8,  9, 10, 11]])`
- `>>>`
- `>>> a[b1]                                     # same thing`
- `array([[ 4,  5,  6,  7],`
- `       [ 8,  9, 10, 11]])`

Advanced slicing Techniques

```
>>>
>>> a[:,b2]                                  # selecting columns
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10]])
>>>
>>> a[b1,b2]                                 # a weird thing to do
array([ 4, 10])
```

The ix() function

- The ix_ function can be used to combine different vectors so as to obtain the result for each n-uplet. For example, if you want to compute all the a+b*c for all the triplets taken from each of the vectors a, b and c:

```
•    >>> a = np.array([2,3,4,5])
•    >>> b = np.array([8,5,4])
•    >>> c = np.array([5,4,6,8,3])
•    >>> ax,bx,cx = np.ix_(a,b,c)
•    >>> ax
•    array([[[2]],

•           [[3]],

•           [[4]],

•           [[5]]])
•    >>> bx
•    array([[[8],
•            [5],
•            [4]]])
```

The ix() function

- >>> cx
- array([[[5, 4, 6, 8, 3]]])
- >>> ax.shape, bx.shape, cx.shape
- ((4, 1, 1), (1, 3, 1), (1, 1, 5))
- >>> result = ax+bx*cx

The ix() function

```
>>> result
array([[[42, 34, 50, 66, 26],
        [27, 22, 32, 42, 17],
        [22, 18, 26, 34, 14]],

       [[43, 35, 51, 67, 27],
        [28, 23, 33, 43, 18],
        [23, 19, 27, 35, 15]],

       [[44, 36, 52, 68, 28],
        [29, 24, 34, 44, 19],
        [24, 20, 28, 36, 16]],

       [[45, 37, 53, 69, 29],
        [30, 25, 35, 45, 20],
        [25, 21, 29, 37, 17]]])
>>> result[3,2,4]
17
>>> a[3]+b[2]*c[4]
17
```

Linear Algebra

```
>>> import numpy as np
>>> a = np.array([[1.0, 2.0], [3.0, 4.0]])
>>> print(a)
[[1. 2.]
 [3. 4.]]

>>> a.transpose()
array([[1., 3.],
       [2., 4.]])

>>> np.linalg.inv(a)
array([[-2. ,  1. ],
       [ 1.5, -0.5]])
```

Linear Algebra

```
>>> u = np.eye(2) # unit 2x2 matrix; "eye" represents "I"
>>> u
array([[1., 0.],
       [0., 1.]])
>>> j = np.array([[0.0, -1.0], [1.0, 0.0]])

>>> j @ j        # matrix product
array([[-1.,  0.],
       [ 0., -1.]])

>>> np.trace(u)  # trace
2.0
```

Linear Algebra

- >>> y = np.array([[5.], [7.]])
- >>> np.linalg.solve(a, y)
- array([[-3.],
-         [4.]])

- >>> np.linalg.eig(j)
- (array([0.+1.j, 0.-1.j]), array([[0.70710678+0.j
  , 0.70710678-0.j   ],
-         [0.   -0.70710678j, 0.   +0.70710678j]]))

Automatic Reshaping

- To change the dimensions of an array, you can omit one of the sizes which will then be deduced automatically:

Automatic Reshaping

```
•   >>> a = np.arange(30)
•   >>> b = a.reshape((2, -1, 3))  # -1 means "whatever is needed"
•   >>> b.shape
•   (2, 5, 3)
•   >>> b
•   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]]])
```

Vector Stacking

- How do we construct a 2D array from a list of equally-sized row vectors? In MATLAB this is quite easy: if x and y are two vectors of the same length you only need do m=[x;y]. In NumPy this works via the functions column_stack, dstack, hstack and vstack, depending on the dimension in which the stacking is to be done. For example:

Vector Stacking

- `>>> x = np.arange(0,10,2)`
- `>>> y = np.arange(5)`
- `>>> m = np.vstack([x,y])`
- `>>> m`
- `array([[0, 2, 4, 6, 8],`
- `       [0, 1, 2, 3, 4]])`
- `>>> xy = np.hstack([x,y])`
- `>>> xy`
- `array([0, 2, 4, 6, 8, 0, 1, 2, 3, 4])`

Building histograms with numpy

- The NumPy histogram function applied to an array returns a pair of vectors: the histogram of the array and a vector of the bin edges. Beware: matplotlib also has a function to build histograms (called hist, as in Matlab) that differs from the one in NumPy. The main difference is that pylab.hist plots the histogram automatically, while numpy.histogram only generates the data.

Building histograms

- import numpy as np
- rg = np.random.default_rng(1)
- import matplotlib.pyplot as plt
- # Build a vector of 10000 normal deviates with variance 0.5^2 and mean 2
- mu, sigma = 2, 0.5
- v = rg.normal(mu,sigma,10000)
- # Plot a normalized histogram with 50 bins
- plt.hist(v, bins=50, density=1)   # matplotlib version (plot)
- # Compute the histogram with numpy and then plot it
- (n, bins) = np.histogram(v, bins=50, density=True)
- # NumPy version (no plot)
- plt.plot(.5*(bins[1:]+bins[:-1]), n)