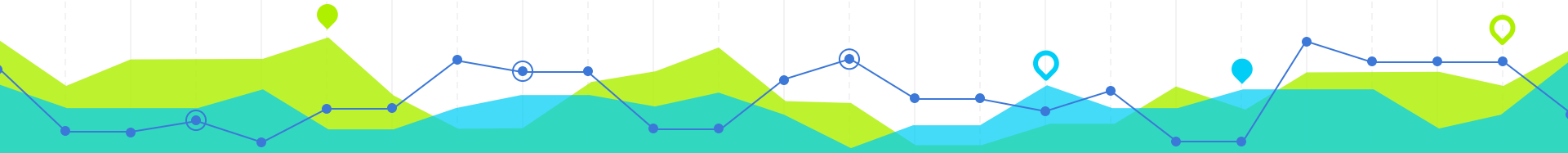
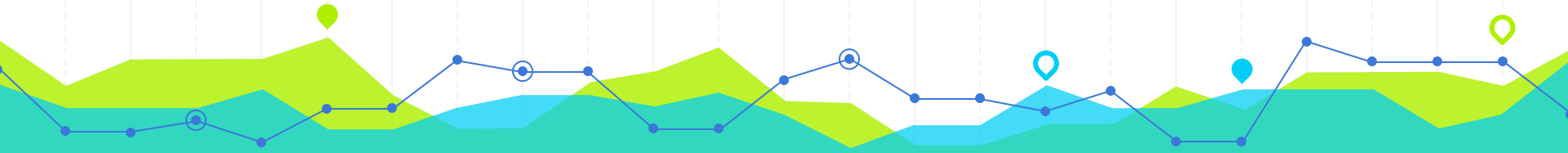
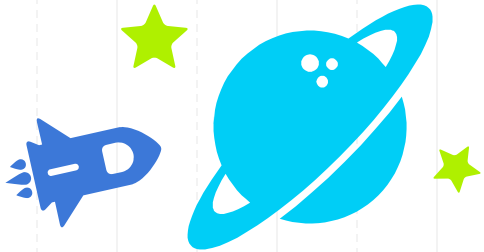


What is Artificial Intelligence?

What is Machine Learning?



What is Data science?



NUMPY

This package is required to play with 2d arrays in python.

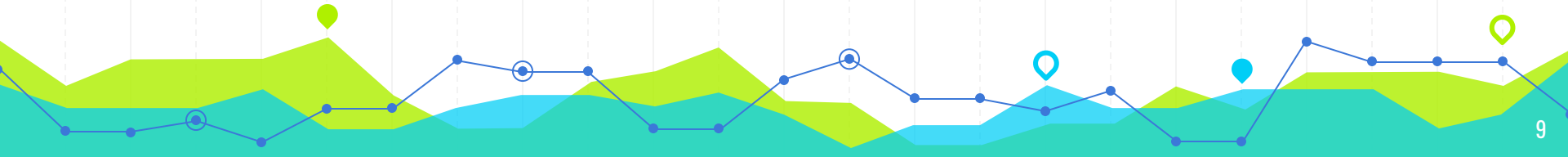
It’s faster than list and easy to understand.

NUMPY OPERATIONS

- Basic Operations
- Array Creation
- Printing Arrays
- Universal Functions
- Indexing, Slicing and Iterating
- Shape Manipulation
- Stacking different arrays
- Splitting Arrays
- Copies and Views

“

Import numpy as np



Basic Operation

```
>>> 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]])
```

Basic Operation

- **Ndarray.ndim**

```
>>> a.ndim
2
```

- **Ndarray.shape**

```
>>> a.shape
(3, 5)
```

Basic Operation

- **Ndarray.size**

```
>>> a.size
15
```

- **Ndarray.dtype**

```
>>> a.dtype
'int64'
```

Basic Operation

- **Ndarray.itemsize**

```
>>> a.itemsize
8
```

- **Ndarray.data**

```
>>> a.data
(Full Array)
```

```
>>> type(a)
<type 'numpy.ndarray'>
```

Array Creation

```
>>> 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')
```

Array Creation Syntax

```
>>> a = np.array(1,2,3,4)    # WRONG
>>> a = np.array([1,2,3,4])  # RIGHT
```

2 Dimensional Array

```
>>> b = np.array([(1.5,2,3), (4,5,6)])
>>> b
array([[ 1.5,  2. ,  3. ],
       [ 4. ,  5. ,  6. ]])
```

Array with Special Data Type

```
>>> c = np.array( [ [1,2], [3,4] ], dtype=complex )
>>> c
array([[ 1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j]])
```

Array with all zeros

```
>>> np.zeros( (3,4) )
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
```

Array with all ones

```
>>> 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)
```

Empty Array

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

Empty Array

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

Numpy Range Function

```
>>> 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])
```

Advance Maths Function

```
>>> 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

```
>>> a = np.arange(6)                    # 1d array
>>> print(a)
[0 1 2 3 4 5]
```

Printing Arrays

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

Printing Arrays

```
>>> 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]]]
```

Printing Large Arrays

```
>>> 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]
```

Printing Full Large Arrays

```
>>> np.set_printoptions(threshold='nan')
```

Basic Arithmetic Operations

```
>>> 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])
```

Basic Arithmetic Operations

```
>>> 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], dtype=bool)
```

Basic Arithmetic Operations

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

Basic Arithmetic Operations

```
>>> A.dot(B)                # matrix product
array([[5, 4],
       [3, 4]])
>>> np.dot(A, B)            # another matrix product
array([[5, 4],
       [3, 4]])
```

Basic Arithmetic Operations

```
>>> a = np.ones((2,3), dtype=int)
>>> b = np.random.random((2,3))
>>> a *= 3
>>> a
array([[3, 3, 3],
       [3, 3, 3]])
```

Basic Arithmetic Operations

```
>>> b += a
>>> b
array([[ 3.417022  ,  3.72032449,  3.00011437],
       [ 3.30233257,  3.14675589,  3.09233859]])
```

Unary Operations

```
>>> a = np.random.random((2,3))
>>> a
array([[ 0.18626021,  0.34556073,  0.39676747],
       [ 0.53881673,  0.41919451,  0.6852195 ]])
>>> a.sum()
2.5718191614547998
>>> a.min()
0.18626021137767O9
>>> a.max()
0.6852195003967595
```

Unary Operations with axis attribute

```
>>> 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])
```

Unary Operations with axis attribute

```
>>> 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])
```

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])
```

Universal Functions Addition

```
>>> C = np.array([2., -1., 4.])
>>> np.add(B, C)
array([ 2.,  0.,  6.])
```

Array Indexing

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

Array Slicing

```
>>> a[2:5]
array([ 8, 27, 64])
```

Change alternate element

```
>>> a[:6:2] = -1000    # equivalent to a[0:6:2] = -1000; from start to position 6, exclusive, set every 2nd element to -1000
>>> a
array([-1000,     1, -1000,    27, -1000,   125,   216,   343,   512,   729])
```

Reverse Array

```
>>> a[ : :-1]                          # reversed a
array([  729,   512,   343,   216,   125, -1000,    27, -1000,     1, -1000])
>>> for i in a:
...     print(i**(1/3.))
...
nan
1.0
nan
3.0
nan
```

One Index Per Axis

```
>>> 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]])
```

One Index Per Axis

```
>>> b[2,3]
23
>>> 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])
```

One Index Per Axis

```
>>> b[1:3, : ]                    # each column in the second and third row of b
array([[10, 11, 12, 13],
       [20, 21, 22, 23]])
```

The Last Row

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

The Dots

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

The Dots

```
>>> 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]])
```

Array Iteration

```
>>> for row in b:
...     print(row)
...
[0 1 2 3]
[10 11 12 13]
[20 21 22 23]
[30 31 32 33]
[40 41 42 43]
```

Array Element Iteration

```
>>> for element in b.flat:
...     print(element)
...
0
1
2
3
10
```

Shape Manipulation

```
>>> a = np.floor(10*np.random.random((3,4)))
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.shape
(3, 4)
```

Shape Manipulation - Flat Array

```
>>> a.ravel()  # returns the array, flattened
array([ 2.,  8.,  0.,  6.,  4.,  5.,  1.,  1.,  8.,  9.,  3.,  6.])
```

Shape Manipulation - Reshape Array

```
>>> a.reshape(6,2)  # returns the array with a modified shape
array([[ 2.,  8.],
       [ 0.,  6.],
       [ 4.,  5.],
       [ 1.,  1.],
       [ 8.,  9.],
       [ 3.,  6.]])
```

Shape Manipulation - Transpose Array

```
>>> a.T  # returns the array, transposed
array([[ 2.,  4.,  8.],
       [ 8.,  5.,  9.],
       [ 0.,  1.,  3.],
       [ 6.,  1.,  6.]])
>>> a.T.shape
(4, 3)
>>> a.shape
(3, 4)
```

Shape Manipulation - Resize Array

```
>>> a
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
>>> a.resize((2,6))
>>> a
array([[ 2.,  8.,  0.,  6.,  4.,  5.],
       [ 1.,  1.,  8.,  9.,  3.,  6.]])
```

Shape Manipulation - Auto Calculate Other Dimension

```
>>> a.reshape(3,-1)
array([[ 2.,  8.,  0.,  6.],
       [ 4.,  5.,  1.,  1.],
       [ 8.,  9.,  3.,  6.]])
```

Stacking Arrays

```
>>> a = np.floor(10*np.random.random((2,2)))
>>> a
array([[ 8.,  8.],
       [ 0.,  0.]])
>>> b = np.floor(10*np.random.random((2,2)))
>>> b
array([[ 1.,  8.],
       [ 0.,  4.]])
```

Stacking Arrays - vstack and hstack

```
>>> np.vstack((a,b))
array([[ 8.,  8.],
       [ 0.,  0.],
       [ 1.,  8.],
       [ 0.,  4.]])
>>> np.hstack((a,b))
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Stacking Arrays - Stack 2D Arrays

```
>>> from numpy import newaxis
>>> np.column_stack((a,b))     # with 2D arrays
array([[ 8.,  8.,  1.,  8.],
       [ 0.,  0.,  0.,  4.]])
```

Stacking Arrays - Stack 2D Arrays

```
>>> a = np.array([4.,2.])
>>> b = np.array([3.,8.])
>>> np.column_stack((a,b))     # returns a 2D array
array([[ 4., 3.],
       [ 2., 8.]])
>>> np.hstack((a,b))          # the result is different
array([ 4., 2., 3., 8.])
```

2D Column Vector

```
>>> a[:,newaxis]          # this allows to have a 2D columns vector
array([[ 4.],
       [ 2.]])
>>> np.column_stack((a[:,newaxis],b[:,newaxis]))
array([[ 4.,  3.],
       [ 2.,  8.]])
```

2D Column Vector

```
>>> np.hstack((a[:,newaxis],b[:,newaxis]))   # the result is the same
array([[ 4.,  3.],
       [ 2.,  8.]])
```

Split Array

```
>>> a = np.floor(10*np.random.random((2,12)))
>>> a
array([[ 9.,  5.,  6.,  3.,  6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 1.,  4.,  9.,  2.,  2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])
```

Split Array

```
>>> np.hsplit(a,3)   # Split a into 3
[array([[ 9.,  5.,  6.,  3.],
       [ 1.,  4.,  9.,  2.]]), array([[ 6.,  8.,  0.,  7.],
       [ 2.,  1.,  0.,  6.]]), array([[ 9.,  7.,  2.,  7.],
       [ 2.,  2.,  4.,  0.]])]
```

Split Array

```
>>> np.hsplit(a,(3,4))   # Split a after the third and the fourth column
[array([[ 9.,  5.,  6.],
       [ 1.,  4.,  9.]]), array([[ 3.],
       [ 2.]]), array([[ 6.,  8.,  0.,  7.,  9.,  7.,  2.,  7.],
       [ 2.,  1.,  0.,  6.,  2.,  2.,  4.,  0.]])]
```

Copy

```
>>> a = np.arange(12)
>>> b = a            # no new object is created
>>> b is a           # a and b are two names for the same ndarray object
True
>>> b.shape = 3,4    # changes the shape of a
>>> a.shape
(3, 4)
```

Shallow Copy

```
>>> c = a.view()
>>> c is a
False
>>> c.base is a                    # c is a view of the data owned by a
True
>>> c.flags.owndata
False
```

Shallow Copy

```
>>> c.shape = 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]])
```

Shallow Copy

```
>>> s = a[ : , 1:3]     # spaces added for clarity; could also be written "s = a[:,1:3]"
>>> s[:] = 10           # s[:] is a view of s. Note the difference between s=10 and s[:]=10
>>> a
array([[   0,   10,   10,    3],
       [1234,   10,   10,    7],
       [   8,   10,   10,   11]])
```

Deep Copy

```
>>> 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

```
>>> 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]])
```

Array Creation Functions

arange, array, copy, empty, empty_like, eye, fromfile, fromfunction, identity, linspace, logspace, mgrid, ogrid, ones, ones_like, r, zeros, zeros_like

Conversions Functions

ndarray.astype, atleast_1d, atleast_2d, atleast_3d, mat

Manipulations Functions

array_split, column_stack, concatenate, diagonal, dsplit, dstack, hsplit, hstack, ndarray.item, newaxis, ravel, repeat, reshape, resize, squeeze, swapaxes, take, transpose, vsplit, vstack

Questions Functions

all, any, nonzero, where

Ordering Functions

argmax, argmin, argsort, max, min, ptp, searchsorted, sort

Operation Functions

choose, compress, cumprod, cumsum, inner, ndarray.fill, imag, prod, put, putmask, real, sum

Advance Numpy - Indexing with Array Indices

```
>>> a = np.arange(12)**2                    # the first 12 square numbers
>>> i = np.array( [ 1,1,3,8,5 ] )           # an array of indices
>>> a[i]                                    # the elements of a at the positions i
array([ 1,  1,  9, 64, 25])
```

Advance Numpy - Indexing with Array Indices

```
>>> j = np.array( [ [ 3, 4], [ 9, 7 ] ] )     # a bidimensional array of indices
>>> a[j]                                      # the same shape as j
array([[ 9, 16],
       [81, 49]])
```

Advance Numpy - Multidimensional vs Single Dimensional

```
>>> palette = np.array( [ [0,0,0],          # black
...                    [255,0,0],        # red
...                    [0,255,0],        # green
...                    [0,0,255],        # blue
...                    [255,255,255] ] )     # white
>>> image = np.array( [ [ 0, 1, 2, 0 ],       # each value corresponds to a color
in the palette
```

Advance Numpy - Multidimensional vs Single Dimensional

```
>>> palette[image]                  # the (2,4,3) color image
array([[[  0,   0,   0],
        [255,   0,   0],
        [  0, 255,   0],
        [  0,   0,   0]],
       [[  0,   0,   0],
        [  0,   0, 255],
        [255, 255, 255],
        [  0,   0,   0]]])
```