Chapter 4. Numpy

What is NumPy?

- a package focus on using arrays for computation
- significantly more efficient (less time-consuming)
- great to handle large datasets
- easy to use, avoid writing loops

Outlines

- **4.1 NumPy ndarray: A Multidimensional Array Object**
  - Creating **ndarray**: N-D array object
  - Data types
  - Arithmetic
  - Indexing & slicing

- **4.2 Universal Functions: Fast Element-Wise Array Functions**
  - ***ufunc*** – a function that performs element-wise operations (a series of functions)

- **4.3 Array-Oriented Programming with Arrays**

4.1 Creating a new array

- Random numbers: np.random.rand(3,2)
- Zeros: zeros(3); zeros((2,3)); zeros_like()
- Ones: ones(2); ones_like()
- Empty: empty((2,3)),empty_like()
- eye,identify

```
np.ones(2)
```

```
array([1., 1.])
```

4.1 Arithmetic with NumPy arrays

- Enable batch operations – vectorization
- Equal-size arrays
- arr * arr
- arr – arr
- 1/arr
- arr ** 0.2
- arr2 > arr

```
In [52]: arr
Out[52]:
array([[ 1.,  2.,  3.],
       [ 4.,  5.,  6.]])
```

- operations between differently sized arrays → broadcasting – Append.

4.1 Basic indexing and slicing

```
In [134]: arr=np.arange(6)

In [135]: arr
Out[135]: array([0, 1, 2, 3, 4, 5])

In [136]: arr2=arr

In [137]: arr2
Out[137]: array([0, 1, 2, 3, 4, 5])

In [138]: arr2[:]=10

In [139]: arr2
Out[139]: array([10, 10, 10, 10, 10, 10])

In [140]: arr
Out[140]: array([10, 10, 10, 10, 10, 10])
```

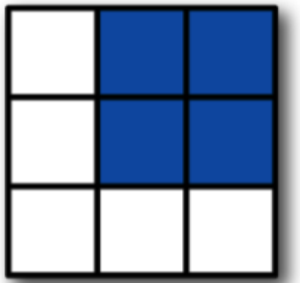
```
>> arr=[0:5]

arr =

     0     1     2     3     4     5

>> arr2=arr

arr2 =

     0     1     2     3     4     5

>> arr2(:)=10

arr2 =

    10    10    10    10    10    10

>> arr

arr =

     0     1     2     3     4     5
```

4.1 Basic indexing and slicing

```
In [141]: arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

In [142]: arr2d
Out[142]: array([[1, 2, 3],
                 [4, 5, 6],
                 [7, 8, 9]])

In [143]: arr2d[2]
Out[143]: array([7, 8, 9])

In [145]: arr2d[0,2]
Out[145]: 3
```

```
>> arr=[1:3;4:6;7:9]

arr =

     1     2     3
     4     5     6
     7     8     9

>> arr(3)

ans =

     7

>> arr(3,:)

ans =

     7     8     9

>> arr(1,3)

ans =

     3
```

```
[47]: arr2d

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



Expression	Shape
arr[:2, 1:]	(2, 2)

```
: arr2d[:2,1:]

: array([[2, 3],
         [5, 6]])
```

```
>> arr(1:2,end-1:end)

ans =

     2     3
     5     6
```

Similarities & differences (Python & MATLAB)

```
data2=np.random.rand(2,3)
```

```
data2
```

```
array([[0.41909519, 0.80645464, 0.06042049],
       [0.49848161, 0.99027677, 0.6950542 ]])
```

```
: data2.dtype
```

```
: dtype('float64')
```

```
: data.shape
```

```
: (2, 3)
```

```
>> data=rand(2,3)

data =

    0.7922    0.6557    0.8491
    0.9595    0.0357    0.9340
```

```
>> class(data)

ans =

double
```

```
>> size(data)

ans =

     2     3
```

Similarities & differences (Python & MATLAB)

```
arr[5:8]
array([5, 6, 7])
```

```
np.ones(3)
```

```
array([1., 1., 1.])
```

```
>> arr=[5:8]

arr =

     5     6     7     8
```

```
>> ones(3)

ans =

     1     1     1
     1     1     1
     1     1     1
```

4.2 Universal functions – ufunc

- simple element-wise ufuncs – *unary* ufuncs
  - np.sqrt()
  - exp()
- operations that take two arrays and return a single array – *binary* ufuncs
  - maximum(x,y)
  - add()
- returns to multiple arrays
  - mod

```
In [147]: arr
Out[147]: array([-3.2623, -6.0915, -6.663 ,  5.3731,  3.6182,  3.45  ,  5.0077])

In [148]: remainder, whole_part = np.modf(arr)

In [149]: remainder
Out[149]: array([-0.2623, -0.0915, -0.663 ,  0.3731,  0.6182,  0.45  ,  0.0077])

In [150]: whole_part
Out[150]: array([-3., -6., -6.,  5.,  3.,  3.,  5.])
```

4.3 Array-oriented programming

- matrices multiplication

```
In [155]: points = np.arange(-5, 5, 0.01) # 1000 equally spaced points

In [156]: xs, ys = np.meshgrid(points, points)

In [158]: z = np.sqrt(xs ** 2 + ys ** 2)
```

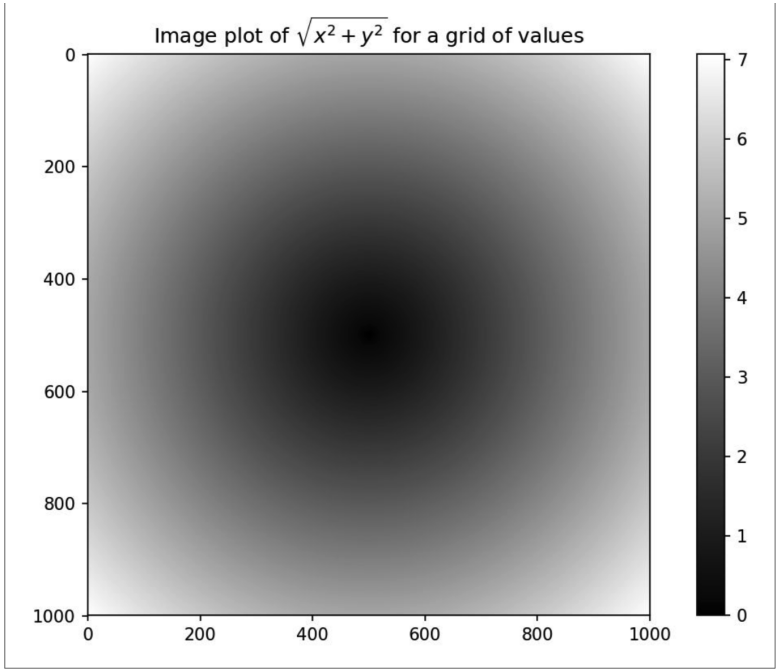


Figure 4-3. Plot of function evaluated on grid

4.3 Array-oriented programming

- meshgrid for matrices with different size

```
In [169]: xs=np.arange(-5,5,1)

In [170]: xs
Out[170]: array([-5, -4, -3, -2, -1,  0,  1,  2,  3,  4])

In [171]: arr2
Out[171]: array([10, 10, 10, 10, 10, 10])
```

xs: 1 x 10
arr2: 1 x 5

```
In [172]: np.meshgrid(xs,arr2)
Out[172]: [array([[-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                  [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                  [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                  [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                  [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4],
                  [-5, -4, -3, -2, -1,  0,  1,  2,  3,  4]]),
           array([[10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10, 10, 10, 10, 10]])]
```

meshgrid for 1x10 and 1x5
→two 5 x 10 matrices

4.3 Array-oriented programming

```
In [173]: np.meshgrid(arr2,xs)
```

```
Out[173]: [array([[10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10],
                  [10, 10, 10, 10, 10, 10]]), array([[-5, -5, -5, -5, -5, -5],
                  [-4, -4, -4, -4, -4, -4],
                  [-3, -3, -3, -3, -3, -3],
                  [-2, -2, -2, -2, -2, -2],
                  [-1, -1, -1, -1, -1, -1],
                  [ 0,  0,  0,  0,  0,  0],
                  [ 1,  1,  1,  1,  1,  1],
                  [ 2,  2,  2,  2,  2,  2],
                  [ 3,  3,  3,  3,  3,  3],
                  [ 4,  4,  4,  4,  4,  4]])]
```

meshgrid for 1x5 and 1x10
→two 10x5 matrices

4.3 Math & Stat methods

```
In [177]: arr = np.random.randn(5, 4)

In [178]: arr
Out[178]:
array([[ 2.1695, -0.1149,  2.0037,  0.0296],
       [ 0.7953,  0.1181, -0.7485,  0.585 ],
       [ 0.1527, -1.5657, -0.5625, -0.0327],
       [-0.929 , -0.4826, -0.0363,  1.0954],
       [ 0.9809, -0.5895,  1.5817, -0.5287]])
```

- calling the array instance

```
In [179]: arr.mean()
Out[179]: 0.19607051119998253

In [182]: arr.mean(axis=1)
Out[182]: array([ 1.022 ,  0.1875, -0.502 , -0.0881,  0.3611])

In [183]: arr.sum(axis=0)
Out[183]: array([ 3.1693, -2.6345,  2.2381,  1.1486])
```

- use NumPy function

```
In [180]: np.mean(arr)
Out[180]: 0.19607051119998253
```

4.6 Pseudorandom number generation – p118

- np.random.normal(size=(4,4))
- np.random.randn(4,4)

normal:

```
numpy.random.normal(loc=0.0, scale=1.0, size=None)
# Draw random samples from a normal (Gaussian) distribution.

# Parameters :
# loc : float -- Mean (“centre”) of the distribution.
# scale : float -- Standard deviation (spread or “width”) of the distributio
# size : tuple of ints -- Output shape. If the given shape is, e.g., (m, n,
```

So in this case, you're generating a ***GENERIC*** normal distribution (more details on what that means later).

randn:

```
numpy.random.randn(d0, d1, ..., dn)
# Return a sample (or samples) from the “standard normal” distribution.

# Parameters :
# d0, d1, ..., dn : int, optional -- The dimensions of the returned array, s
# Returns :
# Z : ndarray or float -- A (d0, d1, ..., dn)-shaped array of floating-point
```

In this case, you're generating a ***SPECIFIC*** normal distribution, the standard distribution.

https://stackoverflow.com/questions/21738383/python-difference-between-randn-and-normal